U.S. Department of Commerce

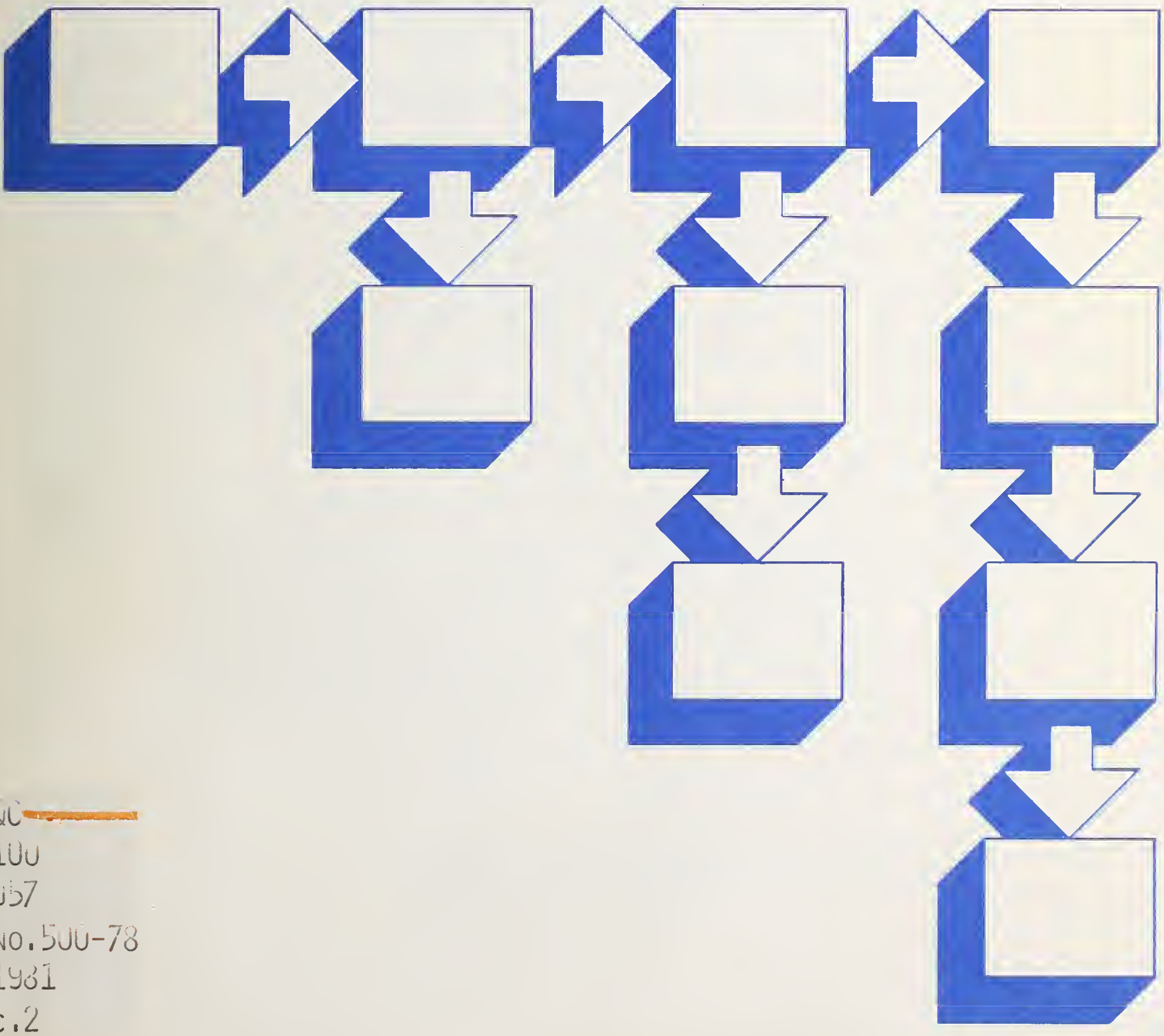
National Bureau of Standards

Computer Science and Technology

NBS PUBLICATIONS

NBS Special Publication 500-78

NBS Programming Environment Workshop Report



Computer Science and Technology

NBS Special Publication 500-78

NBS Programming Environment Workshop Report

Editors:

Martha A. Branstad
W. Richards Adrion

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

[illegible]

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued June 1981

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-78
Nat. Bur. Stand. (U.S.), Spec. Publ. 500-78, 106 pages (June 1981)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 81-600068

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1981

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
Price $4.75
(Add 25 percent for other than U.S. mailing)

TABLE OF CONTENTS

NBS PROGRAMMING ENVIRONMENT WORKSHOP REPORT

edited by

Martha A. Branstad
W. Richards Adrion

In May of 1980, NBS hosted a workshop to assess the state-of-the-art in programming environment technology and to determine the key questions and issues that must be addressed to use these techniques to improve software quality and productivity within the Federal Government. This document reports the results of the workshop.

Key words: development support systems; programming environments; software development; software tools; toolboxes.

EXECUTIVE SUMMARY

To help the ICST staff assess the possibilities in the present and future use of computers to increase both software quality and productivity, a small workshop for invited participants was held 29 April through 2 May, 1980, at Rancho Sante Fe, CA. The participants were asked to investigate various approaches to automating software development including the use of "integrated tool systems" and "high level language environments." Both of these approaches are often referred to as programming environments. The workshop was divided into four working groups:

* contemporary software development environments,
* software environment research,
* advanced development support systems, and
* high level language environments.

As might be expected in a new area of high interest, the workshop participants did not always agree. However, there was consensus on three major items:

* additional automation should be used to assist software development,

* successful information management is a critical but perplexing issue, and

* experimentation and prototype construction are needed to investigate concepts.

The group tasked with analyzing the contemporary systems developed the most definitive statement. It defined four development environments, each more comprehensive and automated than its predecessor. The most modest development environment, for medium size projects, augmented basic system tools with:

* a manual requirements definition and specification methodology,
* a data dictionary to facilitate design,
* an automated (but simple) source code control tool,
* a file comparator for use in verification, and
* manual milestone charts to support project management.

The two groups looking into the future proposed quite a range of possibilities. The area of most fundamental disagreement involved what could be accomplished within a 5 to 10 year timeframe. This disagreement was based upon very different perceptions of the current state-of-the-art. Some saw present day reality in terms of projects underway in their research labs; others viewed reality as mirrored by the technology used in production shops today. The profound lag in technology transfer from construction of research prototypes to the use of the concept in software production was emphasized by these divergent viewpoints. Participants also strongly disagreed on the intended user group for future software development environments.

The disparity of views about the user population naturally led to differences of opinion about the functions required within a development environment. Surprisingly, the technical approaches proposed had a strong thread of consensus. Most participants viewed software development as a series of refinements of objects from the general requirement specification to the concrete realization of the program. The important research lies with discovering the transformations and increasingly automating application of the transformation. The rapid construction of prototypes was viewed as an important component of future development environments and an important concept to apply to the development of the environments themselves. Usable and efficient environments will be created only through continued environment construction, experimentation, and reconstruction. Almost everyone saw data management as the heart of a software development environment. However, few techniques or approaches were proposed to deal with the wealth of information that all felt should be kept. The participants agreed that development support systems and programming environments were important research areas.

CHAPTER 1 INTRODUCTION

Introduction

The Institute for Computer Science and Technology within the National Bureau of Standards carries out the following responsibilities under P.L. 89-3Ø6 (Brooks' Act) :

* develops Federal automated data processing standards,
* provides agencies with technical assistance for ADP, and
* undertakes necessary research in computer science and technology.

The goal of P.L. 89-3Ø6 is the "economic and efficient purchase, lease, maintenance, operation, and utilization of automatic data processing equipment by Federal agencies." As part of its current standards initiative, ICST is studying methods to ensure the quality of software developed for the government. Central to these efforts is the desire to find new and automated means for supporting and enforcing disciplined software development. The production of high quality software developed at a reasonable cost in a timely fashion continues to be an elusive goal. Hiring trained personnel to produce the software has become an increasingly difficult task as professional computer scientists have become a scarce commodity. Thus the need to increase both the quality of the software being produced and the productivity of each computer scientist has become an ever more pressing concern. The solution appears to lie in the use of the very computer technology we aim to serve.

Automation must be used to serve and augment itself. An increasing portion of the software development process should be computerized. This can be accomplished by providing automated tools to aid the software developer. Many tools already exist. Compilers, text editors, file comparators, program analyzers, and test harnesses are examples of such tools. However, rarely have these tools been designed to work in concert. Consequently, the assistance they provide is often awkward and incomplete. The pieces must be joined together to create a whole more powerful than the parts. A programming environment is the embodiment of this concept. It provides automated assistance at each stage of the development cycle to augment the potential of each computer scientist both in the quality and the quantity of what is produced.

Historical Perspective

Tools to aid the programming process were among the first programs written, at least in the modern age of computing. Programs could not be loaded, assembled, and executed without loaders, assembers, device drivers, and other rudimentary tools. Without these simple aids, the programmer was left to manually "patch" or key in data. Almost immediately, tools were made more sophisticated and designed to help the programmer code and debug programs. These included compilers, cross referencers, run-time instrumentation tools, debuggers, text editors, and system libraries. Tools and languages proliferated.

Just as coding was previously viewed as the key software task, tools to support the coding phase were the most prevalent. As the emphasis has shifted and software engineering has stressed the total lifecycle, interest has increased in tools to support each phase and the development process as a whole. Increased emphasis upon visible products throughout the process, has led to increased use of "formal", processable languages for expressing requirements and designs. These languages, in turn, have provided a basis upon which analysis can be done and tools to perform such functions have been built. An increase in interactive computing has also influenced the desire to have a well engineered set of tools to support software development. All these factors have led to an increased interest in programming environments to support the complete software development process.

The task of developing such support environments is quite difficult. The software development process, itself, is still not completely understood, with languages and techniques for use in the early lifecycle stages undergoing significant investigation. Although specific tools to support individual portions of the lifecycle development process exist, joining the tools together into a coherent and useful system is often extremely difficult.

Other issues complicate the problem further. Researchers frequently must contend with a disenchantment with software tools in general. Many factors have led to this:

1. Tool developers have made extravagant and unjustified claims for their products. Many tools have been oversold as a panacea, while they provide only modest functions.

2. Many tools have been developed in research and academic environments as demonstrations of concepts. Though successful as prototypes they were never engineered as production products, though they are being used as such.These tools are usually not well documented, not efficiently coded, not robust enough for a production

environment, seldom portable, and never adequately supported. Consequently they are unsatisfactory to use.

3. Tools are not often designed with integration in mind. Where interfaces exist, they are rarely simple and clean. Consequently, using several tools in concert is usually difficult (if not impossible).

4. The proliferation of languages and language versions, subsets, and supersets has made tool development difficult. Tool builders have too often opted to build tools too tightly coupled to the implementation language, the application language, the hardware environment and/or the operating system. These constraints cannot be completely avoided, but portability, extensibility, and ease of use are important factors in tool acceptance.

5. Tools have been designed for experts. Many tools are poorly human engineered for the novice or even the journeyman programmer. The cry for "user friendly" systems is heard far and wide.

6. Little analysis of the benefits of automation of software development has been undertaken. Managers have been asked to accept that tools increase productivity, reduce errors, provide more useful documentation, ease maintenance, etc., without reliable data to back up these claims. Cost-benefit analysis in this area is very difficult, but is sorely needed.

Benefits from Programming Environments

A well engineered set of tools that is easy to use and which work together smoothly offer many advantages, both to the manager and to the software developer. As with any set of tools well suited to the task, the software support environment can make the work easier and the final product better. Proper tools can increase the productivity of the worker. With a programming environment that supports the complete lifecycle, intermediate products such as requirement and design specifications are produced. These products help to make progress more visible and give the manager an opportunity to have better control over the project. Programming environments encourage and support record keeping which in turn promotes the maintainability of the final system. Visible intermediate products and complete development records also provide a basis for more insightful and comprehensive contract monitoring. Information is available so developers, managers, and purchasers can determine progress and product usability. Programming environments, development support systems, integrated tool systems, whatever the name, the concept of integrated, automated support for software development appears to offer a great potential for improving software quality and productivity.

Workshop Goals

To help the ICST staff assess the possibilities in the present and future use of computers to increase both software quality and productivity, a small workshop for invited participants was held 29 April through 2 May, 1980, at Rancho Sante Fe, CA. The participants were asked to investigate and discuss the use of two related but distinct approaches to automated software development, an integrated tools system and a language environment approach. The integrated tools systems can be further subdivided into the toolbox and the development support system approaches. A development support system is a collection of individual tools appropriately interfaced, with a user front end and an underlying database. With a toolbox, the ensemble of tools, the tool application, and the tool output must be more directly managed by the user. A comprehensive collection of these tools available in a user engineered system should increase software quality and productivity dramatically. There are difficult questions to be solved before such a system could be operational. Among these are the need to identify a core set of tool features, to integrate the tools into a system, to provide flexibility and expandability of service, and to support data management .

An alternate approach to development is through the use of a high level language environment in which code is evolved through successive refinement of the initial specifications. This approach involves the direct automation of the development process by embedding tool features in a high level language itself. The language environment would allow the programmer to specify the problem at a very abstract level, and then gradually evolve the final code by successive refinement. Interpreters, syntax-directed editors, consistency checkers, correctness verifiers, and compilers are tools that would be incorporated into such an environment. Powerful data management techniques built on knowledge-based structures would be required. Such a language environment is at the edge of current research in high level languages, operating systems, data structures, artificial intelligence, database management, and program verification.

Both the tool approach and the language environment have been called "programming environments"; they are complementary with the integrated tool approach being nearer term. Both approaches were the subject of the NBS workshop. The goals of the workshop were to provide an assessment of the current technology, an indication of needed standards for software tools, guidance for developing practical near term development support systems, suggestions for assessing the impact of such technology, and research directions in programming environments and software development. The workshop was to advise NBS on how to direct its efforts to achieve high quality software and increased productivity for the Federal Government through the use of automation during the software development process.

Workshop Organization

In order to provide manageable topics and discussion, the subject and participants were organized into four groups. Following considerable debate, the final partitioning of the subject matter was based upon a time continuum. Group 1, Contemporary Software Development Environments, led by Dr. William Howden was tasked with looking at what could be done today, the most near term approach to development environments. Dr. Leon Osterweil led Group 2, Tool Integration Strategies. This group was to consider what could be produced using today's technology but with a five year delivery date. The remaining two groups were tasked with looking at research issues. Group 3, Advanced Support Systems, chaired by Dr. Thomas Standish was tasked with providing a framework for critical research and development issues. Group 4, Language Environments, was to consider development environments built around and in support of a programming language. Dr. Marvin Zelkowitz chaired Group 4. The personalities and interests of the participants and the group interaction caused a somewhat different path to be taken in some of the groups. Group 1 focused clearly on the near-term and set forth concrete recommendations while Group 2 concentrated on short-term research questions. Group 4 considered language issues, while Group 3 addressed itself to the farthest reaches of the research time continuum.

Report Organization

This report mirrors the organization of the workshop itself. The next four chapters present the deliberations of each of the four working groups. Summaries for each and for the workshop as a whole follow in Chapter 6. In order to more clearly delineate the domain of each working group and to stimulate thought prior to the workshop, questions were generated for each group. These sets of questions and issues appear in Appendix B. Workshop participants are listed in Appendix A.

CONTEMPORARY SOFTWARE DEVELOPMENT ENVIRONMENTS

William Howden

Participants
Paul Cohen, Defense Communications Agency
William Howden, University of Victoria
Al Irvine, Softech
James King, IBM
Patricia Powell, National Bureau of Standards
William Riddle, University of Colorado
Leon Stucki, Boeing Computer Services
Leonard Tripp, Boeing Computer Services

1.Ø Introduction

A. Software Development Environments.

The software development environment consists of the methods, techniques, and tools which are used during the development of a software system. Early environments consisted of a compiler and a linking loader. Later environs included editors and debuggers, informal requirements and design methods, and simple programming standards. Many new methods and software tools have been formulated and built during the last decade and the software development environment has evolved into a rich structure of development technology. It is estimated, for example, that there are now more than 4ØØ commercially available software development tools.

In May of 198Ø, the U.S. National Bureau of Standards sponsored a workshop on software development environments at Rancho Sante Fe, California. One of the several groups at the workshop was assigned the task of studying near-term environments. These are environments that could be built within the next two or three years using state-of-the-art technology. The model for the environments that were considered was that of a toolbox: a collection of automated tools and methods that could be used to build quality programs. The toolbox group was successful in designing a succession of increasingly complex environments. It also discussed the problems of tool and method integration and a simple powerful approach to this problem was proposed. This report describes the basic features of the tool environments and the approach to integration.

B. Software Development Products.

The approach that was followed in the design of the tool environments was to consider first the products that must be generated by a software project and then the methods and tools that would be useful in generating those products. This was an effective approach. It provided structure and direction to the efforts and avoided the pitfall of attempting to prejudge the usefulness of a specific tool or tool function without first discussing the need for it.

The software life cycle was used as the software development model during the discussion of the products that are built during a software development project. The life cycle model emphasized the importance of both intermediate and final products. The needs of different classes of personnel were also considered. Analysts, programmers, customers, managers, and operations staff are responsible for and depend on the use of different development documents. Some of the more important products are listed below. Some products are closely associated with particular phases of the life cycle and others transcend the phases.

REQUIREMENTS PRODUCTS

Two major classes of requirements products were identified. The first consists of requirements definition documents. These documents are generated during the definition of requirements and form the initial communication medium with the customer. A special language such as SADT [31,32] may be involved. The second product consists of the requirements specification. This is a formal contractual document that defines the system which is to be built. It is constructed after requirements definition and, ideally, is represented in some formal language or graphical notation. PSA/PSL [38] can be thought of as a requirements specification language.

Requirements definition documents are often informal. They may consist of large pieces of paper tacked up on the walls, constructed in a scissors and paste mode. Graphical languages such as SADT [32] can be used during requirements definition to build requirement's models. Requirements specifications are constructed after the definitional phase has been completed when the requirements can be stated more formally and in more detail. A prototype user manual for a system may also be a product of the requirements phase. A test plan based on the functional properties of the system which are described in the requirements specifications was also considered to be an important requirements product. It should define both test data and expected results as well as procedures for running the tests.

DESIGN PRODUCTS

"As-built" design representations are geared to help users understand and maintain a system. The program logic manual for the IBM 370 is an example of an "as-built" product. "As-built" representations may contain less detail than "build-to" development design representations which are used by the system development staff. Structured design diagrams [40] can be used to construct examples of "build-to" design representations. A design-based test plan that is based on the system functions introduced at the design stage should also be generated [16]. The design of a system is often divided into preliminary (or architectural) design and detailed design. The preliminary design may be used to construct a build plan. A build plan describes the order in which modules or parts of the system are to be designed and built. The preliminary design may also be used to construct schedules, budgets, resources management procedures, milestone charts, and maintenance documents. A maintenance document may describe the "design envelope" for a system. It lists the kinds and scope of the alterations and customizing which are possible without altering the design.

CODING PRODUCTS

In addition to the source and object code modules, the coding phase involves the construction or modification of management products (budgets, schedules, etc.), user manuals, change control plans (discrepancy reports, procedures for correcting errors), and code based test plans. The coding phase also involves the generation of the reports of all testing and code analysis activities that are carried out.

MAINTENANCE

Products which are important to and are either used with or generated during maintenance include configuration specifications, change control procedures and plans (including regression tests, data and results), cross reference documents, and all requirements and design specifications.

C. Classes of Software Projects.

Different toolbox environments are appropriate for different classes of projects. Two types of projects were considered. The first was a medium sized automatic data processing application and the second a large embedded real time software system.

The medium sized project was assumed to have the characteristics listed in Table 1. The list is incomplete and is meant only to delimit certain basic properties of a class of projects. The description of the users as sophisticated means that they are capable of understanding and evaluating preliminary design specifications. The support staff for the project includes user representatives, industrial engineering personnel who carry out management and planning functions, separate testing staff, and clerical, and operations personnel.

Development time	2 years
Project budget	$2,000,000
System lifetime	15-20 years
Development staff	7 programmers & 1 manager (plus support staff)
Users	Sophisticated
Area	ADP

Basic characteristics of medium size project

TABLE 1

Basic characteristics of the large system are listed in Table 2. The development of large systems having these properties introduces a whole new set of problems. One of the most significant is communication between development personnel. Many of the issues which must be considered for large systems but not medium sized result from what are basically system engineering and management problems.

Development time	3-5 years
Project budget	$20,000,000 (development only, no maintenance)
System lifetime	10 years or more
Development staff	70 programmers 5-7 managers
Users	Unsophisticated
Area	Embedded real time

Basic characteristics of large scale project

TABLE 2

The number of programmers on the large project will vary and may start with as few as twenty. The budget of $20,000,000 does not include maintenance which, over the life of the system, may amount to $60,000,000. In addition to the characteristics in Table 2, it was assumed that the large scale project could involve different development and implementation machines, geographical dispersion of development personnel, and that it would be part of an essentially new application.

D. Classes of tool environments.

Three different classes of tool environments were designed for medium sized projects and two for large projects. The first of the three environments for medium sized projects, code named Fig leaf, covers the bare essentials. It was considered to contain the minimum set of tools and methods without which it would be foolish to attempt to carry out such a project. The second environment, code named Leopard skin, covered most of the areas in which development staff need environmental support. It contained tools and methods for assisting the user in all of the more important parts of the development process. The third environment, code named Overalls, provides a complete and workmanlike coverage of all tool and technique areas.

Two classes of environments were discussed for large scale systems. The first was essentially the same as the second environment for medium systems. Leopard skin was considered to be the minimal tools and techniques environment for the development of a large scale system. The second environment for large scale systems, code named Spacesuit, is an elaboration of Overalls.

A rough estimate of capital cost for each of the environments was made on the assumption that most of the tools would be purchased and not developed in-house. The cost estimates provide some idea of the relative cost for each environment but it was difficult to feel confident with these figures as absolute. It is reasonable to assume that the costs are measured in units that range somewhere from an American dollar to a British pound. It was assumed that the training and recurring costs for tools are amortized over many projects and are not included in capital costs. Capital cost estimates were made by estimating what different kinds of tools might sell for.

E. Assumed tools and techniques.

The "assumed tools" are the traditional tools that are part of all software development environments. They include compilers, link-editors, assemblers, run-time support routines, and, in some cases, a source code debugging system. The assumed tool set also contains a filing system with a built-in back-up and recovery mechanism. The assumed tools for large-scale systems include cross-compilers, simulators, and emulators. In some projects the assumed tools may have special features. It may be necessary, for example, to have a cross-compiler that is capable of generating code for either the development or the target machine. It is likely that some of the assumed tools in the large scale system environments will be customized for the application. None of the assumed tools are discussed in later parts of the report. The primary goal is to discuss tools and techniques which are part of the current state-of-the-art for software development environments and which can be added to the assumed traditional set of development tools.

2.Ø Fig leaf

a) CAPITAL COST: $35,ØØØ

b) REQUIREMENTS TOOLS AND TECHNIQUES. Fig leaf contains no automated requirements tools. Requirements definition and specification are carried out manually. It is assumed that some systematized, but unsupported (by tools) methodology such as data flow diagrams[12] or HIPO charts[35] will be used for requirements definition and specifications.

c) DESIGN TOOLS AND TECHNIQUES. The use of a data dictionary tool for ADP projects was considered essential [11]. Other aspects of the design were assumed to be manual. On a medium sized project the single manager was assumed to be the designer. The problem of design is greatly simplified if there are no communication problems. The generation of functional design documents was considered essential although no particular design methodology is suggested. The approach to design that is followed in Fig leaf is a function of the manager's experience and the nature of the application problem. Functional design documents should describe the functions introduced at the design stage of the software project. They describe functions of individual modules and parts of programs and are essential to the generation of test data [15,16].

d) CODING TOOLS AND TECHNIQUES. Fig leaf contains a simple automated source code control tool [2,3Ø]. The tool is built on top of a conventional text manager for filing and retrieving blocks of text. The source code control tool allows the programmer to enter different versions of procedures and modules. The tool keeps an index of and allows retrieval of different versions of the objects. The source code control system may have facilities for automatically retrieving older versions of a module or procedure.

e) VERIFICATION TOOLS AND TECHNIQUES. The only automated verification tool included in Fig leaf is a file comparator [8,1Ø]. The comparator can be used for different purposes including version management and regression testing. No other automated verification tool is included. Verification in Fig leaf is largely manual. Test plans and files of test data are constructed and managed manually. The test plans describe functions or system features to be tested. They are based on information from requirements and design documents as well as specific features of the code. Test plans may be indexed and associated with different versions of modules or procedures or different configurations of the system.

f) MANAGEMENT TOOLS AND TECHNIQUES. No automated management tools are included in Fig leaf. It is assumed that manual milestones or Gantt charts will be maintained to support management activities.

3.Ø Leopard skin

a) CAPITAL COST: $2ØØ,ØØØ.

b) DATA BASE. Leopard skin is assumed to include a simple software engineering data base for storing and retrieving products that are built during a software project [19,38]. It is assumed that the source code control system and cross-reference tools work off the data base. Other tools and techniques in Leopard skin may or may not use the data base. The software engineering data base in Leopard skin is built around three concepts: that of an object, an object property, and a relationship between objects. Objects may be code modules, test plans, design documents, milestone charts, requirements specifications, or any other software development product. The objects may be textual or internal representations of graphical objects. Any kind of property can be defined for an object or any kind of inter-object relationship.

c) DEPENDENCY ANALYSIS. One of the major tacks in software engineering is the management of dependencies between objects. Examples of dependencies include the "compiled" relationship between a source code module and an object code module, the "calls" relationship between a calling and a called procedure, the "implements" relationship between a design function and piece of code, the "must precede" relationship between two tasks in a build plan, data coupling relationships between two modules in a structured design, and the "part of" relationship between a configuration specification and a set of modules and files. The property and relationship features of the software engineering data base can be used to support tools for entering and analyzing dependencies between related objects.

Leopard skin contains a dependency analysis tool for entering dependency relationships and for retrieving objects that are related by named dependencies. Sophisticated general purpose dependency analyzers are capable of carrying out more complex tasks such as finding the transitive closure of a relationship.

Dependency relationships are entered into the data base in two ways. The first is manually. The managers decide which kinds of dependencies are to be maintained and then guidelines and audit procedures are set up to ensure that the dependencies are entered and maintained. Dependency relationships may also be entered into the system automatically, either as a side effect or through the use of special purpose dependency analysis tools.

Compilers, for example, can be constructed so that they automatically enter call relationships between procedures. This is an example of a side-effect. A module cross-reference tool can be built whose only purpose is to analyze inter-module references and enter the appropriate relationships into the data base.

It is important not to confuse special purpose dependency analysis tools and report generators with the simple, general purpose dependency analyzer that interacts directly with the data base. The special purpose tools are built on top of the general purpose analyzer. It is also possible to build special purpose tools which report on dependencies which are not stored in the data base. In this case the dependency analyzer examines objects directly for certain properties and reports on the dependencies of interest without entering them into the data base. The only assumed dependency tool in Leopard skin is the simple general purpose tool that is used by other special purpose tools.

d) REQUIREMENTS TOOLS AND TECHNIQUES. Leopard skin contains tools for storing and manipulating machine readable requirements specifications. Requirements definition may continue to be a manual methodology. The tool for entering specifications is, like the source code entry system, a special kind of editor that is "knowledgeable" about the specification language. It may know for example, that if "A uses B", then "B is used by A". The requirements specification tool is not required to use the software engineering data base. It may use ordinary files. If it uses the data base then it may be built so that it automatically causes the entry of dependency relationships such as "A uses B" (and the consequent derived "B is used by A") into the data base [38]. Requirements definition and specification may involve requirements languages or both languages and methodologies for defining or specifying requirements. PSA/PSL, for example, is primarily a specification language. SADT involves both a language (graphical) and a comprehensive requirements methodology. Other examples of requirements definition and specification methodologies include HIPO [35], SAMM [36] and SREM [1].

e) DESIGN TOOLS AND TECHNIQUES. The use of a formal, machine readable design representation is assumed in Leopard skin. Possible design methods include the Jackson method [2Ø], PDL [7], TOPD [34], structured design [4Ø], module descriptions [4], data dictionaries [11], HOS [15], SARA [9], and Nassi-Schneiderman diagrams [23]. Tools must be available both for entering and retrieving designs and for carrying out associated design analysis. Design analysis tools can be used, for example, to generate PDL cross-references, execute paths, and show all possible data flows in a TOPD design, or check the consistency of interfaces in a module structure.

f) CODING TOOLS AND TECHNIQUES. Leopard skin contains more elaborate source code entry tools than Fig leaf and it is assumed that the tools are built to use the software engineering data base. Source code modules may have many properties associated with them such as the time when the module was entered, required hardware, required operating system, and dependencies on other modules (some combinations of modules go together and some don't).

Configurations consist of collections of particular versions of modules or collections of modules that have some other specified property. Leopard skin allows users to define objects called configurations and contains a configuration management tool for retrieving the modules belonging to a particular configuration. The configuration manager may need to call on other tools in order to form a configuration and must therefore be compatible with these tools. The manager may find, for example, that a module which is to be included in some configuration has not been compiled, in which case it will call the compiler to generate an object module for the source module. The configuration manager may allow different ways of specifying configurations (e.g., latest versions of modules, versions for certain hardware, etc.).

g) VERIFICATION TOOLS AND TECHNIQUES. In addition to the file comparator included in Fig leaf, Leopard skin includes several other automated verification tools. A test coverage analyzer [28,37] is assumed to be available as well as a test harness (or testbed) [25]. Test plans are required to be stored in machine readable form and to contain descriptions of the code to be tested, input variable values, and expected output variable values. The test harness should be able to automatically test sections of code specified in test plans and be able to check the agreement of computed against expected output. Test harnesses should be capable of testing parts of procedures as well as whole procedures and modules. Test plans may specify stubs to be used for procedure calls.

h) MANAGEMENT TOOLS AND TECHNIQUES. Leopard skin contains an automated project control system. The system maintains a model of the software development task. The model may vary in complexity from a milestone or Gantt chart to a CPM or PERT network. The project control system tools implement facilities for storing management information and for generating reports. A Gantt chart project control system, for example, will contain facilities for allowing time cards to be filed against the Gantt task/personnel matrix. It will also allow the generation of reports which summarize time and dollar costs for particular tasks.

i) ACCESSORIES. There is a wide variety of other software engineering tools available. Leopard skin is not expected to contain any more than a small number of these, say two or three. The accessories are expected to be "off-the-shelf" type products. Possible Leopard skin accessories (beads and feathers) include those listed in Table 3.

data flow analyzer [24]
pretty printer (source code)
flow charter [2]
control flow analyzer (generate call graphs) [5]
interface checker (for procedure and functions)
performance monitor (time spent on code sections) [27,29]
module cross-reference tool (uses data base)
documents cross-referencer
units checker (for weakly typed programming languages)
documents preparation tool (for generating reports) [22]
test data generator
Accessory Tools

Table 3

4.Ø Overalls

a) CAPITAL COSTS: $3ØØ,ØØØ.

b) TOOLS AND TECHNIQUES. Overalls contains all of the tools and techniques in Leopard skin, including all accessories.

c) USE OF DATA BASE. As many of the tools and methods as possible store the objects which they manipulate and generate in the software engineering data base.

d) TOOL COMPATIBILITY. Tools which do not use the data base must be compatible. There are several criteria for determining compatibility. Compatible tools should be able to call one another. The configuration manager, for example, must be able to call the compiler to compile a source module. Sources and destinations for the input and output for a tool must be parameterized. In order to be able to use tools together, it must be possible to tell them where to get their data and where to put their output. A cross-reference tool, for example, must not be built in such a way that it always prints its ouput on the line-printer, and is not capable of sending the information to other destinations.

5.Ø Spacesuit.

a) CAPITAL COST: $3,ØØØ,ØØØ.

b) DATA BASE FACILITIES. Space suit contains the same software engineering data base as overalls, as well as several additional information management facilities. All tools and techniques are assumed to use the data base.

Space suit contains an archiving facility for storing previous versions of requirements, design, and source code documents. It is not unusual in a large project to forget why different requirements and design decisions were made. The availability of an archive system allows project personnel to examine previous versions before altering some document to make sure that they are not literally going in circles and re-introducing a feature that was part of a previous, abandoned version.

There are often one or two people on a project who are knowledgeable about what is going on in all parts of a design or in different modules in the system under development. They function as a sort of technical Ann Landers, providing advice about the use of common facilities in the system. The position is self perpetuating, each time a programmer comes for advice he probably gives an advisor more information than he gets in return. In time the advisor will have an enormous knowledge of the system.

The function of the advisor can be partially filled by an indexing tool which is capable of searching the data base for objects which have certain properties. In a generalized data base management system the function would be fulfilled by the query language. Care must be taken to prevent use of the indexer to violate the principle of information hiding. The indexer should not be used to allow a programmer working on one module to find out about and then start rummaging around in the data maintained in another module which is being worked on by some other programmer. The primary purpose of the indexer is to answer questions like "Is there a sort facility mentioned in any of the design documents?"

Policies must be used during the development of a large system for controlling changes to configurations and for controlling the generation of new versions of different kinds of documents. It is especially important that the policies be rigorously enforced in a multi-user environment. Partial control can be maintained by including a facility in the Space suit data base that monitors all changes to objects [3Ø]. Each time a user attempts to file away an object which has been constructed by changing a previous version of the object, the change control facility requires that he give a reason for the change. Different change logs are maintained for different classes of documents. The change control system may also implement a policy for creating new versions of objects and new system configurations.

In addition to the archiver, indexer, and the change control tools, Space suit is assumed to include a sophisticated cross-reference facility for cross-referencing the enormous volume of documents (requirements, design, code, etc.) that is produced on a large project.

c) REQUIREMENTS TOOLS AND TECHNIQUES. The use of an automated requirements specifications tool like that used in Overalls is assumed. The tool, and the associated specifications language, must allow the incremental construction of specifications [14]. Both the requirements definition and specification methodology must incorporate procedures for incremental construction and review. The use of formal, systematic requirements reviews is considered essential.

The need for incremental build and review procedures for large scale systems is partially due to the size of the requirements documents. It is not acceptable to construct and then deliver a 5ØØ page specification to a customer and then ask him for his opinion. The delivery of a hierarchically organized document, a chapter at a time after the entire document has been completed, is not acceptable either. Examination of the first few levels of specifications may reveal a problem that could wipe out the rest of the document.

The requirements definition and specification stage may involve experiments in which part of the requirements are actually carried through to an executable part of the system, or a simulation of that part of the system. The user interface for an interactive system, for example, may be simulated to allow the user to get a feel for the proposed specification for that part of the system.

d) DESIGN TOOLS AND TECHNIQUES. Space suit, like Overalls, includes an automated design system. Design representations (textual and/or graphic) are stored in the data base. Design simulation tools such as SIMSCRIPT and GPSS may also be included. The design methodology in Space suit, like the requirements methodology, must allow phased or incremental construction. The methodology must contain formal procedures for design reviews [39]. Formal reviews are less important in smaller projects where the designers (there may be only one) can easily communicate with each other.

Incremental or phased design is necessary in a large project because different parts of the design are done by people who cannot keep in constant communication with each other. Parts of the design may be sent out to different geographical locations for development. At certain stages the design will go through a consolidation or synthesis phase in which several chief designers will go through the whole design and then possibly take it apart and put it back together again. One of the problems that the design synthesis activity will help solve is that of redundancy. Lower level parts of a design that are being developed by different teams may be 85 per cent the same, unknown to both teams. With some restructuring of the design it may be possible to remove the redundancy and to design a significantly smaller and cleaner system.

e) CODING AND VERIFICATION TOOLS AND TECHNIQUES. Space suit contains essentially the same coding and verification tools and techniques as Overalls.

f) MANAGEMENT TOOLS AND TECHNIQUES. All of the management tools and techniques from the smaller environments are included in Space suit, together with new features that are used to help deal with the complexity introduced by the increased size of the projects that are being developed. Space suit includes methods for constructing build plans and tools for supporting their use [33]. Build plans are based on preliminary (or architectural) designs. They specify the order of construction of different parts of the design. There are a number of reasons why different parts of a system may have to be developed in a specified order. One part may depend on another for testing. Some parts may have a high payoff and can actually be put into use before the other parts. It may be necessary to carry high risk parts to completion before the design and implementation of other parts. It may also be necessary to build prototypes of some pieces of a system. The manager for a large scale project may decide that dependencies between build plans and preliminary designs should be stored in the data base. Related dependencies between build plans and project control budgeting and scheduling information may also be stored in the data base. A Space suit management tool can be built which can be used to analyze the dependencies and to record the effect of a change in the design of the build plan. Space suit may involve industrial engineering staff for project monitoring. Space suit contains a mail box facility for communication between different development personnel.

6.Ø Tool Integration

One of the most common arguments against tool integration is that it imposes an inflexible development methodology upon the development staff. One imagines an enormous Rube Goldberg machine consisting of a nest of hooked up levers, pipes, and wheels. In order to use tool A you must first use B and the output from A is always processed by C and then D unless there is input from E.

The idea that an integrated tool environment must consist of a complex structure of interconnected tools is misleading. The fundamental feature of some of the more famous software support systems is not interconnections but the use of a common kind of data object by all tools or facilities in the system. The properties of the basic data object and the knowledge that different parts of the system have of objects of that type characterized the degree and kind of integration that is found in the system. The UNIX operating system, for example, uses the file as a common basic data object. Little is assumed about files and the kind of integration found in UNIX is relatively bland or general purpose. INTERLISP is built around the use of list structures (S-expressions) as the basic data object. There are a number of well defined kinds of list structures (functions, A-lists, prop-lists, etc.) which have interesting properties that are known about by different parts of the INTERLISP sytem.

The object/property/relationship software engineering data base which is included in all but the simplest of the environments described in this paper can be used to build an integrated software development environment. The data base provides an integrating and unifying medium for interfacing tools without forcing them into a complex structure of inter-relationships. Tools get their information from the data base and put their results back into it without having to interface directly with other tools. The data base can be used to gain the advantage of integration without losing the flexibility of the toolbox.

The advantages of an integrated tool system which uses a common data base include that of eliminating the necessity for multiple copies of the same information. The existence of different copies of the same information for different tools creates a horrible consistency and synchronization problem. Every time one copy or version of a collection of information is updated or changed, all other copies must be changed also. The expense and tedium of doing this virtually rules out the practicality of using certain collections of tools unless they can be modified to work off the data base. Tools which satisfy the requirements for tool compatibility (parameterization of input/output locations and the ability to call one another) can often be attached to a software engineering data base with interface or "data translator" routines. In order to maintain flexibility it is important to avoid building bridges between pairs of tools rather than bridging them into the data base.

Properties other than the use of a common data base have been suggested for an integrated tool system. One is the use of a homogeneous command language for controlling and using the tools [13]. There are mixed opinions on this idea. It may help the user by enforcing uniformity at the interface. Alternatively, it may hinder the user. There are different levels of human interfaces for different tools and it may be difficult to construct a concise, non-redundant uniform command language for all tools.

7.Ø Tool Standards

Two approaches to standardization of environments were discussed by the toolbox group at the NBS workshop. The first was that of standardization of tool function. This is appropriate for tools which have been around for some time and whose basic features have stabilized. It is not appropriate for many other tools, even though their use and capabilities are well understood. Standardization of this type may have a frustrating and inhibiting effect on software development. It may also be simply ignored. Another approach to standardization is that of setting standards for the software environment data base. Standard formats for "input from" and "output into" the data base would retain benefits of standardization such as portability without inhibiting the development of new tools and methods.

8.Ø Tool Effectiveness and Impact

It is one thing to propose a class of software development environments. It is another to convince people to use one of them. The arguments that the use of a tool environment results in the construction of better systems at lower cost, that it gives increased management visiblity and increased analytic ability to the designer and programmer, must be supported with mathematical or empirical evidence. Studies have been caried out on individual methods and tools [e.g. 17] which indicate that they can be powerful, effective devices but there is little empirical data on the use of complete software development environments like Leopard skin, Overalls, and Space suit.

The use of a tool or toolbox environment is more likely to occur if it is human factor engineered. Built-in dynamic (step through examples) tutorials are one way of doing this [26]. Getting into and out of a tool must be simple. One way of measuring the quality of the human factors engineering in a tool is to see how far you can get with the tool by guessing how to control and use it before you need the user manual.

Perhaps the most direct way of encouraging the development and use of a software engineering environment is to require that the software development contractor be formally obligated to produce the intermediate as well as the final products that are associated with the software life cycle. Required intermediate products may include, for example, data flow diagrams, data dictionaries, data flow analysis reports, and project status reports. The software development contractor who is uninterested in using the tools and techniques in a modern development environment may be the same contractor who does not really believe that the life cycle and its associated intermediate products are really necessary. If these products are formally required it should be possible to convince the contractor that his job will be easier if he uses a tool environment which is designed to manage and make the production of these products not only easier but, in some cases, possible.

9.Ø Summary

Four classes of software development environments have been proposed. Each can be built in the near term using state-of-the-art technology. The environments were designed by first considering the life cycle products that are generated during a software development project. The environments contain tools and techniques which can be used to construct those products.

The products oriented approach to software environments reveals the existence of many activities which are carried out during a project which otherwise may not be thought of. The formal recognition of the existence of all of the life cycle products which are generated during a software development project may end up making the development time longer but the final product should be better and the collected intermediate products will make effective maintenance possible.

The basic unifying component in the three more sophisticated of the four classes of environments is the software engineering data base. It is predicted that the use of a simple data base will not only make integration possible, but it will make the construction of an integrated environment easier than that of an unintegrated toolbox.

Two classes of tools and techniques were excluded from the proposed environments: those beyond the current state-of-the-art and those whose range of applicability is highly restricted or whose cost/effectiveness is still open to question. The first includes, for example, transformational programming systems [3] and the second, symbolic evaluators [18] and program mutation systems [6]. The references for the included tools and techniques indicate typical examples and are not meant to be complete.

The use of a software engineering environment is encouraged if contractors are formally obliged to produce intermediate life cycle products. The effectiveness of individual tools has been studied but there is little information available on the effectiveness of complete environments. A research project which included the construction of and an evaluation of the effectiveness of a software development environment would provide important and much needed information about the question.

1Ø.Ø References

[1] M.W. Alford, "A Requirements Engineering Methodology for Real-time Processing Requirements," IEEE Transactions on Software Engineering, Vol SE-3, 1977.

[2] Autoflow II Users Guide, Applied Data Research, Princeton, New Jersey.

[3] R. Balzer, N. Goldman and D. Wile, "On the Transformational Implementation Approach to Programming," Proceedings 2nd International Conference on Software Engineering, Los Angeles, 1976.

[4] B. Boehm, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," IEEE Transactions on Software Engineering, Vol SE-1, 1975.

[5] J. Brown, "Getting Better Software Cheaper and Quicker," in E. Horowitz, Editor, Practical Strategies for Developing Large Software Systems, Addison-Wesley, Reading, Mass. 1975.

[6] T.A. Budd, R.J. Lipton, and G. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," Proceeding National Computer Conference, AFIPS, 1978.

[7] S.H. Caine and K.E. Gordon, "PDL-A Tool for Software Design," National Computer Conference Proceedings, Vol 44, 1975.

[8] E.D. Callender, "Industrial Practices to Control Computer Program Costs," SAMSO-TR-75-293, 1976.

[9] I.M. Campos and G. Estrin, "Concurrent Software System Design Supported by SARA at the Age of One," Proceedings 3rd International Conference on Software Engineering, Atlanta, 1978.

[1Ø] Compare Facility, Network Operating System, CYBER Series, Control Data Corporation.

[11] Data Manager Users Guide, MSP Inc., Lexington, Mass.

[12] T. DeMarco, Structured Analysis and System Specification, Yourdon Inc., N.Y., 1978.

[13] J.R. Ehrman, "The New Tower of Babel," Datamation, Feb., 198Ø.

[14] D.P. Freedman and G. Weinberg, Ethnotechnical Review Guide, Ethnotech, Lincoln, Nebraska, 1979.

[15] M. Hamilton and S. Zelden, "Higher Order Software-A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, 1976.

[16] W.E. Howden, "Functional Testing and Design Abstractions," Journal of Systems and Software, (to appear).

[17] W.E. Howden, "Applicability of Software Validation Techniques to Scientific Programs," ACM Transactions on Programming Languages and Systems, July, 1980.

[18] W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Transactions on Software Engineering, Vol. SE-3, 1977.

[19] C.A. Irvine and J.W. Brackett, "A System for Software Engineering," Infotech State-of-the-Art Report on Structured Analysis and Design, Infotech, Maidenhead, England, 1978.

[20] M. Jackson, Principles of Program Design, Academic Press, London, 1975.

[21] D.B. Knudsen, A. Berfsky, and L.R. Satz, "A Modification Request Control System," Proceeding 2nd International Conference on Software Engineering, Los Angeles, 1976.

[22] Librarian User Reference Manual, Applied Data Research, Princeton, New Jersey.

[23] I. Nassi and B. Schneiderman, "Flowchart Techniques for Structured Programming," ACM Sigplan Notices, Vol. 8, 1973.

[24] L.J. Osterweil and L.D. Fosdick, "DAVE- A Validation Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, 6,1976.

[25] D.J. Panzl, "Automatic Software Test Drivers," Computer, 11,1978.

[26] R.L. Patrick, "A Checklist for System Design," Datamation, January, 1980.

[27] Program Problem Evaluator (PPE) Users Guide, Boole and Babbage, #U-D503, Palo Alto, California.

[28] C.V. Ramamorthy and S.F. Ho, "Testing Large Software with Automated Evaluation Systems," IEEE Transactions on Software Engineering, Vol. SE-1, 1975.

[29] Resolve Users Manual, Boole and Babbage, Palo Alto, California.

[30] M.J. Rochkind, "The Source Code Control System," IEEE Transactions on Software Engineering, Vol SE-1,4,1975.

[31] D.T. Ross and K.E. Schooman, "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, SE-3, January, 1977.

[32] SADT, The Softech Approach to System Development, Softech, January, 1976.

[33] D.J. Schultz, "A Case Study in System Integration Using the Build Approach," Proceedings ACM National Conference, 1979.

[34] R.A. Snowden and P. Henderson, "The TOPD System for Computer-aided System Development," Infotech State-of-the-Art Report on Structured Analysis and Design, Infotech, Maidenhead, England, 1978.

[35] J.F. Stay, "HIPO and Integrated System Design," IBM Systems Journal, 2,pp 143-154.

[36] S.A. Stephens and L.L. Tripp, "Requirements Expression and Verification Aid," Proceedings 3rd International Conference on Software Engineering, Atlanta, 1978.

[37] L.G. Stucki, "New Directions in Automated Tools for Improving Software Quality," in R. T. Yeh, Editor, Current Trends in Programming Methodology, Vol. 2, Prentice-Hall, Engelwood Cliffs, N.J., 1978.

[38] D. Teichroew and E.A.Hershey, "PSA/PSL: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, 1, 1977.

[39] E. Yourdon, Structured Walkthroughs, Yourdon Inc., New York, 1977.

[40] E. Yourdon and L. Constantine, Structured Design, Prentice-Hall, Englewood Cliffs, N.J., 1979.

SOFTWARE ENVIRONMENT RESEARCH---THE NEXT FIVE YEARS

Leon Osterweil

Participants

Lori Clarke, University of Massachusetts
Donald Good, University of Texas
Raymond Houghton, National Bureau of Standards
Thomas Love, ITT
Leon Osterweil, University of Colorado
Patricia Santoni, NOSC
Daniel Teichroew, University of Michigan
Athony Wasserman, UC-San Francisco

1.Ø Philosophical Background

1.1 Nature of Software Environments.

In our early deliberations about how to design a five-year research plan for studying software environments, it became increasingly clear that we would have to address the central question of what a software environment really is. We were reticent to attempt a formal rigorous definition, believing that this might lead to unproductive squabbling over wording, and also that this research area is still too young and speculative to lend itself to such formalism. Believing, nevertheless, that this is a central issue, we opted to reach a consensus about the nature of software environments through construction and discussion of examples.

Our discussions of examples spanned the following software scenarios to be supported by environments. This abbreviated list is intended to be representative and is by no means exhaustive.

* A large group (5ØØ) of programmers is charged with the maintenance of long lifecycle (1Ø years) software which they, in general, have not written themselves. This group needs (at least) tools to help study and analyze the code; maintain version control over the code; and test and verify changed code.

* A team of real-time programmers is in the early stages of analysis and design for a distributed real-time processing system. The team needs tools to facilitate the process of understanding and agreeing upon the system requirements, creating, adjusting and verifying the system

design, coordinating and controlling the system coding and testing, and assisting in the process of altering system code, design and/or requirements whenever this is indicated.

* A management team is responsible for the software production efforts of up to several dozen people, possibly working on several different projects. The team needs to make crucial resource allocation decisions in order to assure timely and acceptable completion of the projects. The team needs tools and mechanisms for obtaining visibility into the activities and progress of the software production personnel and projects.

* An independent team for quality control analysis is charged with the responsibility of determining the presence or absence of errors from finished software products in a timely and cost-effective fashion. It needs a collection of analysis and testing capabilities for proceeding in a straightforward way to an identification of specific errors, or for raising, in an orderly, systematic way, its confidence in the absence of errors.

In each of these scenarios it is clear that individual tools and aids do exist to facilitate the work, but that these tend to be isolated, stand-alone tools capable of supporting only isolated parts of the work to be done. What is needed, however, are tool configurations to provide integrated, continuous support. Hence, we came to agree that in a general, vague sense an environment should be thought of as a software system which attempts to redress the failings of individual software tools through serendipitous integration.

It is necessary to elaborate on this statement by being more specific about the nature of the "failings" of individual tools and what is meant by "serendipitous integration." Each of the four scenarios just presented can be relatively easily described by a word or phrase characterizing the nature of the underlying software job to be done. Thus the first is a maintenance scenario, the second a large-scale production scenario, the third a management scenario, and the fourth a Quality Assurance (QA) or Verification and Validation (V&V) scenario. An environment is characterized and distinguished by the fact that its specific mission is to help individuals and teams to perform their software jobs more effectively. Insofar as a support system's tools, interfaces and internal structures are integrated to provide strong, comfortable, continuous support for a specific job at hand, the system can be considered to be "serendipitously" integrated to meet the needs of that job, and hence to be an environment. Individual tools "fail" because their support, although often strong, is usually only narrowly focused on isolated aspects of a software job.

It now becomes clear that the task of creating effective environments is elusive and difficult because it is tantamount to understanding the nature of the fundamental software processes themselves. A specific environment does not merit the name unless it provides strong uniform support for the entire process it is intended to facilitate. That is not possible until and unless that process is fully appreciated and understood. Hence in a fundamental sense, environment development seems destined to progress more or less in parallel with our growing understanding of software processes.

Presently we seem to be in a good position to make substantial progress along these parallel lines. We have grasped the critical importance of software development, maintenance, management and quality assurance. We have begun to differentiate between them and yet also to see their close interrelations. We have constructed significant individual tool capabilities in these areas and gained valuable experience (both good and bad) with them. Partially through these experiences we have come to better understand the underlying processes which are in need of support. We have defined procedures and methodologies to guide humans in these various software endeavors, and have begun to gain experience with them. In summary, we seem poised to attempt to meld together the most promising of our tools into systems of support for our most promising methodologies in the key areas of software production, management, maintenance, and V&V.

Although optimistic about current prospects for making significant progress in this work, we were also struck by the immensity of it. One of our group (Donald Good) advanced the opinion that this work is at least as ambitious and difficult as the problem of creating superior high level programming languages. In terms of progress, Good believes we are at a very preliminary stage corresponding to the period immediately prior to the appearance of Fortran in the mid-195Ø's. The analogy seems quite apt. What lies ahead for us is a long period of development and experimentation during which we must match automated tools and problem expression media and notation with developing understanding of underlying problem areas and emerging solution methodologies. It seems clear that this matching process will proceed through a sequence of successive improvements in the form of a succession of environments, much as high level language technology has improved through the creation and evaluation of a succession of high level languages. It seemed agreed, moreover, that the analog of Fortran, i.e. a widely applicable and acceptable, general purpose environment, has yet to appear.

It was also agreed that a vigorous effort towards the goal of producing such an environment is clearly indicated.

1.2 Expected Benefits.

There was a diversity of opinions about the expected benefits of such an environment. Improved software quality, efficiency, manageability, and maintainability all were proposed, but efforts to pinpoint the most overriding and crucial of these bogged down. Here too, the high level language analogy seems useful. It is difficult to determine even now, no less in 1950, the greatest benefit of a high level language. Is it readability, manageability, increased productivity or improved quality? We finally agreed that high quality, general purpose environments will offer ample quantities of all of these benefits. Good suggested that these benefits would probably accrue indirectly, because environments would relieve software people of much drudgery, freeing them for proportionately more creative and intellectual activity. Thus software solutions would be more carefully thought out and arrived at after more numerous and thorough iterations. There seemed to be good agreement about this point.

Thus the group seemed generally agreed upon the obvious desirability of such environments and was eager to press on to discussions of how most effectively to pursue their creation.

It was agreed that this could be done most effectively when guided by a considerable base of diverse knowledge, much of which is currently unavailable. Hence the group next discussed the problem of characterizing and categorizing the knowledge needed.

2.0 Overall Strategy for Our Research Plan

The knowledge base needed for the effective production of general purpose software environments appears to be large and diverse. Hence its accumulation should be guided as much as possible by an overall plan. The purpose of this section is to set up the framework within which such a plan can be developed and articulated.

We decided that the outlines of a research plan could perhaps best be established by first identifying a set of distinguishing characteristics of an environment. Towards this end, we identified five characteristics which we believe a support system must possess in large measure if it is to merit the appellation "environment." In so doing, we are suggesting that the proposed experimental research program be focused on studying these five characteristics: their nature, their achievability, the ways in which they might possibly conflict with each other, and eventually whether they are in fact actually the critical characteristics. The five are:

1. Breadth of Scope and Applicability. An environment must extend strong support to a software person or team across the full range of aspects of the software job being done.

2. User Friendliness. An environment must provide strong, direct, comfortable support. It must not oblige the user to accommodate himself/herself to it, but rather it must accommodate itself to the user. This accommodation must extend beyond the usual items: clear diagnostics, fail-soft error recovery, clear easy-to-use input languages, and HELP subsystems. The environment must in addition provide direct, painless support for the user in the actual procedures of his/her day-to-day work. It must not oblige the user to adapt to or relearn a new way of doing business.

3. Reusability of Internal Components. An environment must be flexible in adjusting to different and changing user needs. This flexibility can probably only be achieved by constructing the environment out of tool fragments, rather than whole tools. We strongly agreed that a collection of monolithic tools, standing side-by-side under the umbrella of a common user interface, is unlikely to be both flexible and efficient in meeting the possibly unforeseen diversity of needs which users may have. A comprehensive collection of easily reconfigurable tool fragments would offer this flexibility with the potential for efficiency as well.

4. Tight Integration of Capabilities. An environment's capabilities must work closely with each other to provide the user with a sense of continuously strong support. An environment must support a user community in doing its work according to its own procedures. Yet the environment itself is to be implemented by a possibly small set of tool fragments to be configured and reconfigured in response to the (possibly changing) requirements of the end-user community. This poses the danger that the user might be made uncomfortably aware of the fact that his/her needs are being met by an amalgam of different tools and tool fragments. This must be strenuously avoided, as it violates the principle of User Friendliness. It can be naturally avoided by assuring that the individual tools and fragments maintain an awareness of the existence and capabilities of each other. Through this awareness the tools, tool fragments, and integrating software should avoid duplication of services and reports. The tools and fragments must also be preconditioned to uncomplainingly tolerate each other's quirks.

5. Use of a Central Information Repository. It was agreed that it is quite reasonable to think of an environment as an information utility. In an important sense, the purpose of an environment is to assure that software workers can get the information they need to do their jobs at the time the information is needed. From this perspective, the purpose of an

environment's tools is to capture such information, analyze and process the information, and disseminate the information. Given this view, we agreed that any environment should be actually implemented along the lines of this model. At the center of the environment must reside a data base of total information about the software project. Surrounding this data base should be an information management system whose job it is to access the data base in response to requests made by the environment's tools, tool fragments and (possibly) user interface components.

It is important to observe here that these five characteristics are not represented to be orthogonal, nor is it suggested that they capture the essence of what an environment ought to be. Hence in that they overlap or conflict, that overlapping or conflict presents a possible obstacle to the eventual effective construction of general purpose environments, and thereby suggests an important area of inquiry and research. Some of these areas of apparent conflict will be identified and discussed in some detail in subsequent sections.

3.Ø What Must Be Learned from the Research Plan

This section considers each of the five distinguishing characteristics of an environment. For each the nature and importance of the characteristic is elaborated, with respect to the subject of software environments. We then explain the central questions which must be explored in order for true general purpose software environments to become realities.

3.1 Breadth of scope and applicability.

The central issue here is the need to determine just how broad and encompassing an environment can reasonably be expected to be. One member of our group observed, only half in jest, "there must be something like 2 raised to 1Ø to the 6th different environments," that might be built. These differ from each other along a multitude of coordinates which one might use for categorization, and, indeed, in the coordinates which are appropriate.

The representative list of examples of support systems, given in the first section of this report, begins to indicate this diversity. That list indicated that it is reasonable to consider building environments to support software development, maintenance, verification, testing and management. Other software activities worth considering are documentation and distribution.

Within each of these activities there is considerable diversity in what might be supported. For example a software production environment might have to support any particular software lifecycle model or concept. Thus it might encompass some or all of: requirement analysis, preliminary (architectural) design, detailed (algorithmic) design, and coding. It would presumably also support some level of testing, analysis and verification. This support might be applicable only to the output of the coding phase or to any or all other phases. The environment would have to generate reports, summaries and analyses upon which to base the various reviews called for by the lifecycle model as well. Similar broad variation should be expected among all support systems which might be considered environments for facilitating the various other software jobs.

More variation must be expected as a result of differences in source languages. A verification or maintenance environment for COBOL programs must of necessity be different from one for LISP programs. There is a certain amount of obvious truth to that statement. The issues can become much deeper, however, when one considers the remarkable range of programming languages and the attendant effects they have on support environments. A language such as LISP is different from COBOL or FORTRAN in some very fundamental ways. Some of these differences make it possible, indeed natural, to edit, test and analyze LISP programs in elegant and powerful ways which would be impossible for a language like COBOL. The INTERLISP system, for example,[7] exploits this, giving a tangible example of the profound impact that a language can have on its support environment.

In a similar way, we noted that an environment to support EL-1 [2] program production would have certain fundamental differences from environments for most other contemporary languages. El-1 supports the design phase, as well as the coding phase, of software development. Hence testing and certain verification procedures can and should be applied uniformly to designs and code in an EL-1 development environment (as is currently being done [3]). The close confederation of these phases, on the other hand, makes it more difficult to separate and identify progress on these phases. This could complicate matters in the creation of an EL-1 management environment.

Different application areas must inevitably also lead to differences in the environments needed to support them. For example, FORTRAN might be used to create a numerical software library or a spacecraft control system. In the former case there would typically be little or no formal requirements analysis and preliminary design. This appears to be due to the maturity of the problem area and the suitability of mathematics as a requirements and design notation. In the latter case there would be extensive amounts of requirements and design creation, analysis, review and reporting. Clearly the environment's support mechanisms would have to vary similarly. The latter problem area is also generally considered to be of more critical importance than the former, as errors have the clear potential for causing loss of life and property. Thus a testing and verification environment for spacecraft control would of necessity include costly verifiers and simulators which would probably be inappropriate in a numerical software testing and verification environment. Indeed, because spacecraft control software is concurrent, tools for testing, analyzing and verifying the concurrent behavior of this software would be essential here, but of no value for the numerical library. Different problem areas also mandate the need for differences in security mechanisms, version controls, and customer reporting in environments supporting these problem areas.

A project's size can also have an important impact on the support environment for that project. Here the primary effect seems to be the need for better and more effective communication and control as project size grows. The communication needs of a 2-3 person project are obviously far more modest than the needs for a 1ØØ-person project, involving 2-3 levels of management. In the larger project there are also needs for privacy and configuration management and control which are either absent or sharply reduced in a small project environment.

Having thus established that there is a need for an enormous number of different environments we are now left with the question of how to supply them all.

Some questions which seem worth exploring as vehicles for elucidating this overriding question include the following:

* Under what circumstances, if any, is it reasonable to synthesize larger, more encompassing environments out of smaller ones?

* Along what degrees of freedom (if any) can we expect to transform one environment for use in meeting a related set of needs. (e.g. it seems reasonable to build tool modules which could be altered to change a PL/I production environment to a FORTRAN production environment. What other sorts of alterations can we expect to be able to make?)

* What sorts and amounts of methodological change in a using project can be comfortably supported and adapated to by a support environment?

Although a certain amount of this inquiry seems self-contained, the answers to these questions must certainly come, at least in part, out of the research into the other four characteristics of an environment, to be described next.

3.2 User Friendliness.

As noted earlier in this report, an environment must present its repertoire of support capabilities to its users in as supportive, unobtrusive and non-interfering a way as possible. There are a few ramifications of this basic requirement that bear elaboration. Most fundamentally, the underlying tool capabilities must be robust enough to survive user abuse (intentional or inadvertent), communicative enough to both explain errors in use and instruct in proper use, and liberal enough to both accept user input and produce user output in a form and language close to the form and language of the software activity being supported. Individual software tools invariably suffer disuse and distrust for lack of one or more of these essential characteristics. Hence it is all the more important that constituent tool capabilities all be robust, communicative and colloquial.

User friendliness in an environment, however, entails more than just assuring the friendliness of individual tools. In addition the accessibility and usability of the entire package of tools must also be assured. Thus, for example, there must be adequate mechanisms for acquainting the user with the range of capabilities available for guiding him/her to the selection of appropriate capabilities. This need to keep an accurate catalog of available capabilities seems clear. What is less clear is whether the catalog should be used as a basis for attempting to reduce duplication of capabilities. Experience suggests that a sort of Software Darwinism tends to cause better capabilities to automatically supplant weaker capabilities without the need for external enforcement. On the other hand, overly large, confusing ensembles of tools can present an intimidating hostile appearance to the user that could discourage the use of an environment.

It was agreed that, regardless of the size or sophistication of any tool cataloging or indexing scheme, it is, nevertheless, essential that the scheme, and the underlying capabilities themselves, be able to communicate with the user in a way with which the user is familiar and comfortable. The central issue here is that the purpose of an environment is to support the user in performing his/her job. This communication between the user and the environment must be in the terminology of the user's job setting. Support capabilities extended must furthermore directly support the methodologies and institutions of the user's job setting, rather than forcing the user to alter his/her way of doing or thinking about his/her work in order to use the environment capabilities.

In some sense what is being described here is not simply friendliness to the user, but rather friendliness to the user's way of doing work. This appears to be more easily demanded than furnished. As already observed, if we are to be saved from having to recreate every support system and environment from scratch, it will be necessary to configure environments as much as possible from standard modular capabilities. We now recognize that this configuration must be done in such a way as to directly support the user's way of getting his/her job done, no matter what that may be. The user's procedures should, in addition, be expected to change with time. The support environment must likewise change accordingly while remaining supportive and friendly to the new procedures. This appears to require the use of extremely flexible, robust, and compatible modular capabilities. It will be necessary to determine whether it is reasonable to expect to be able to build such modules which, nevertheless, display acceptable efficiency characteristics.

Finally, the group explored the issue of whether artifical intelligence and human factors research might be applicable to investigations of user friendliness in environments. The group seemed agreed on this in principle, but considered at length only the narrower issue of whether computer graphics might prove useful in helping to achieve friendly user interfaces. It was agreed that graphics should be expected to be particularly useful when the problem area and/or its procedures could be naturally captured pictorially. Thus, for example, a software management environment would presumably profit from being able to communicate with its users by way of management procedure diagrams, time and effort graphs, PERT charts, and specimen report forms. Similarly a requirements or design creation aid within a software development environment would presumably profit from being able to directly display the pictorial specifications which are the natural form of SAMM [5], SADT [6] or RSL [1] specifications.

It was less clear whether the use of graphics would be of much help in environments supporting communities where pictorial forms of communication are not already in use. In addition it was suggested that the expense of elaborate graphics systems such as those featuring color and motion will probably only rarely be justified.

3.3 Reusability of Implementation Modules.

Earlier sections of this report have already discussed the apparent need for an environment to be constructed out of small flexibly rearrangeable modules or tool fragments. This appears to be perhaps the only way in which we can expect to construct a number of different environments without having to build each from scratch. It appears to be as basic an idea as the manufacturing notion of building and maintaining a product line (e.g. automobiles, TV sets, appliances) out of a modest set of standard parts and subassemblies. We have also already observed that this notion appears at first glance to complicate efforts at making environments user-procedure-friendly. That goal seems to require the total concealment of the identities and characteristics of the implementation modules, and their smooth welding into a support system patterned closely after the user's own procedures. This appeared to place very heavy demands on the flexibility and interchangeability of these modules, suggesting perhaps that they must each be very narrow in scope if this goal is to be attainable.

The foregoing discussions lead one to believe that the "Internal Reusability" characteristic might not be so much an independent characteristic of environments as, perhaps, a derivative of other characteristics.

Be this considered a derived or independent environment characteristic, there appears to be no doubt that investigation of the feasibility of building environments out of small tool fragments is one of the most pressingly important research areas for the near term. Certainly if experience does not show that significant families of tool fragments can be assembled and found to be highly flexible, broadly applicable, yet acceptably efficient, then it will be necessary to drastically revise our thinking about the practicality of creating a diversity of user-friendly software environments at bearable cost.

We had little difficulty identifying potentially useful tool fragments for use in certain places of some environments. For example, a parser seems to be a good example of a useful tool fragment, as a number of tool capabilities rely upon a facility for creating a token string or parse tree. Thus a single parser would be used by a variety of subsequent tool fragments for doing such things as prettyprinting, error checking, static analysis, or compiling.

The parser would perhaps be coupled with various of these subsequent tool fragments in different environments and at different times. A parser is a particularly good example of a tool fragment, because parsers can be automatically created by parser generators, and hence very readily altered as well, for example, to meet changing needs for different languages and dialects. Hence here is an example of a tool fragment creation mechanism which is very flexible in creating a powerful, reasonably efficient, widely applicable tool component.

This example is encouraging, and it serves to stimulate us to look further for other such tool fragments. It was also proposed that a set of general purpose static analysis modules would constitute a good tool fragment. These modules would implement a small number of widely applicable data flow analysis algorithms, operating only on annotated representations of program data flow. These representations would be created by other tool fragments as abstractions of the original program. Hence the data flow analysis fragment would have little knowledge of extraneous source language detail. Research is showing that a small, well chosen set of data flow algorithms can be useful in error detection, verification, and optimization across a very broad family of source languages. Research also appears to indicate that a concise pseudo-language notation can be used to effectively direct the automatic configuration of the algorithms into the various specific error scanners and verifiers that might be needed in different environments or as the needs of a given environment evolve.

Promising as these examples seem, there was nevertheless a feeling that they are rather isolated examples. Our group was pessimistic about the existence of similar tool fragments to be used in building such important environment components as user interfaces, management reports, generalized editors and graphics packages. It was agreed that such fragments can probably be built, but that they will have to be the products of research and experimentation still to come.

We also agreed that the need for tool fragments to maintain a central data base was perhaps the most pressing need of all. There was some sentiment that acceptable fragments of this sort already exist. Another point of view held that these existing information management system capabilities might prove to be too inefficient to be an acceptable part of a full environment. It was held that the full range of requirements placed on an information management system by an environment were not yet known and must be determined in order to enable definitive study of this crucial area.

3.4 Tight Integration of Tool Capabilities.

We also agreed that a true environment is characterized by the close interaction and cooperation of its constituent capabilities. This issue has already been touched upon in discussing the ramifications of the term "user friendly." In that discussion we stressed that user friendliness specifically implies friendliness to the way in which the user does his/her job. Thus tool capabilities must be merged into a system offering smooth continuous support for the user in the performance of his/her actual work procedures. Clearly this requires at least the appearance that the constituent tool capabilities are working closely together. We felt, moreover, that in a true environment this close cooperation among capabilities must be actual, not simply apparent.

We believe that in a smoothly functioning environment it will be important for the tool capabilities to be aware of each other. Thus tools should facilitate each other's work by pre- and post- processing data structures for each other. Tools should also be careful not to duplicate services and messages to the user. In these ways the efficiency and overall appeal of the environment to the user are assured. Interlisp was prominently mentioned as an example of a support system whose constituent capabilities are very tightly integrated both in fact and appearance, to yield a very appealing system.

We have already discussed the fact that the need for tight integration appears to pose a direct conflict with the need to construct environments out of flexible, general, reconfigurable modules (tool fragments). It seems clear that if the tool fragments are too general and reconfigurable, then they cannot exploit any significant knowledge of each other's workings -- only each other's interfaces. Thus it seems close integration can only be projected as an illusion by such components as the IMS or user interface.

Before resigning ourselves to the inherent irreconcilability of these two characteristics, however, it seems useful to study the Interlisp example. One important motivation cited for needing to build environments out of tool fragments was the need to alter environments to fit a range of (probably evolving) user procedures. We find, however, that Interlisp seems to be able to accommodate itself to a range of (changing) user modes. Thus apparently this can be accomplished with tightly coordinated tool capabilities.

It was argued that Interlisp is still rather narrow in its range of support and user community, thus probably not qualifying as a true environment, and that it is only this restriction in scope that enables it to be both flexible and tightly integrated. The discourse then lapsed into conjecture, culminating eventually in good agreement that much could probably be learned by studying Interlisp and attempting to extrapolate from this example.

3.5 Use of a Central Data Base.

The final, and perhaps most important, characteristic of an environment is that it be coordinated and focussed by access to a central repository of information. It was generally agreed that a software project is profitably thought of as being a coordinated effort to gain and disseminate a highly structured body of knowledge about a problem and its solution. That being the case, the progress of the project will be best assured and facilitated by capturing, structuring and disseminating that body of knowledge as faithfully and effectively as possible. These considerations seem to clearly imply the use of a data base and encompassing information management system as the centerpiece of any environment.

It was agreed that by taking this approach the effective diffusion of knowledge to all project personnel would be facilitated. This certainly does not imply the giving of all pieces of knowledge to all people. Quite to the contrary, effective diffusion of knowledge means purveying to each person precisely that information which he/she needs to accomplish his/her job at any given time. This would be accomplished by using the environment's tools to access the data base for specific data in response to needs as expressed to the tools by users. Tool capabilities might simply search the data base for needed information, might report back combinations of data or data aggregates, might update the data base in response to user input, or might augment the data base with the outcomes of analyses of data base contents as requested implicitly or explicitly by environment users. In all cases the outcome would be an up-to-date, immediately, centrally accessible body of complete project information, whose access would be facilitated by the powerful tools of the environment.

This highly attractive picture seems to us to be marred by serious questions of procedure and practicality. The most immediate questions seemed to us to be questions of what should go into the data base and how it should be organized. The immediate and obvious answer, that "...everything should go into the data base...", was rapidly shown to be inadequate. After discussion and, once again, consideration of examples, it became apparent that there were widely different opinions of what was meant by "everything." For example some people believed that a software production environment should preserve in an archive all obsolete version of code, all discarded designs, even all of the sketches and jottings produced during early problem formulation and conceptualization. Others objected to this, stressing the lack of utility and inevitable large-scale waste of resources inherent in this approach.

A resolution of this discussion appeared to come out of a subsequent discussion of the structure and organization of the data base. Here it was proposed (by D. Teichroew) and widely agreed that the data base must be organized as a model of the software activity being supported. In this approach the users, processes, data items, data flows and procedures of the software activity and setting are modeled and represented as entities; attributes are relations in a data base, managed by an information management system. Thus the data needed by an individual in the performance of his/her work is readily available because it is organizationally grouped together within the data base as the attributes and relations of a small set of entities. The need for analyses and reports can be semi-automatically identified and satisfied as the result of recognizing when entities, attributes, and relations within the data base have no current values. Tools could be invoked (manually or automatically) to supply these values.

Another important concern which was expressed was that we determine what procedures are necessary to insure the correctness and consistency of the data base, especially in the face of the continual changes to which it will be subjected. The magnitude of this problem is perhaps most graphically illustrated by considering the impact on a highly structured software development data base of such an apparently small change as altering a single line of program text. In particular, if this source line is in a declaration statement, then its alteration might render invalid parts or all of such related data base denizens as the token string, parse tree, flow graph, and diagnostic reports. For each change, all possibly impacted data base objects must be known, then analyzed, then perhaps purged or altered. The potential cost of such activities is intimidating. Yet the necessity of these activities is undeniable. It may very well be that consideration of the need and cost to do this sort of updating will be a key factor in determining the size and complexity of data bases for support environments.

4.Ø A Five-Year Research Plan

The purpose of this section is to sketch the outlines of a plan for conducting a research program aimed at providing substantive answers to the questions posed in the previous section. The plan was arrived at after careful consideration of both the knowledge needs, as just described, and the current state of the experience and expertise of the research community. It was noted that most of the learning needed is empirical and pragmatic in nature. A great deal of qualitative experience and quantitative statistics must be accumulated. Further it was noted that a number of researchers are currently poised to begin experimentation aimed at accumulating some of the needed knowledge. Thus, we concluded that it would be most effective for these and other researchers to embark upon a program of experimental work which is guided, at least in a general way, towards the objectives which we have agreed upon as being desirable.

Accordingly, our plan essentially maps out a program of experimentation, having two separate thrusts. The first experimental thrust, intended to commence immediately, calls for the development of prototype support systems, each of which is intended to provide some specific insights into environment characteristics, as well as experience with moderate scale tool integration.

The second experimental thrust, not intended to begin for three years or more, will use the experience gained to attempt the synthesis of some full scale general purpose environments.

The studies of prototype systems during the first thrust should each be aimed at learning about one or more of the five stated characteristics of an environment and their interplays and relationships. It seems to us that the research community is currently in a position to learn a great deal by studying existing support systems and frameworks and also by building a variety of new, small to moderate scale support systems. The study of existing systems is a particularly logical step in that it should provide insights into effective integration strategies as well as answers to questions in some of the five previously described areas. Such studies should also make clear the areas in which more learning is most needed and in which this can be accomplished through new system construction. Thus clearly, this system construction should be done in such a way as to elucidate as many of the outstanding critical questions as possible. Clearly it is possible now to gain a considerable number of important insights into such questions as:

* How to structure and maintain environment data bases?

* Upon what sorts of tool fragments might environments be built?

* What are some specific tradeoffs between flexibility and tight integration of tools?

* What are reasonable uses of graphics in user interfaces?

Following shortly are some suggestions for research projects aimed at providing insights into some of these and related questions. These suggestions are intended to be taken as examples rather than mandates.

It is expected that as this line of experimental research proceeds, the level of tool integration will increase, forming the basis for the second research thrust -- namely large scale integration of tools into general purpose environments. This thrust, though properly based upon the first thrust, is expected to have a different character, focusing mainly on the issue of breadth of scope. It is expected that this line of experimentation will draw upon all previous experience and learning, in attempting to determine the reasonable limits of tool integration. We will certainly find that it is unreasonable to construct an environment so powerful and encompassing as to meet the changing needs of all people at all times. We expect to learn, however, the limits which ought to be placed on the generality and scope of general purpose environments, as well as the techniques which are helpful in achieving large scale tool integration.

Our group seemed agreed that it is important to delay this thrust for 3-5 years, rather than initiate it immediately. There currently seem to be so many important gaps in our knowledge in critical areas, that this sort of enterprise seems inordinately risky. It was felt that any research activity, initiated now, and aimed at such large scale tool integration would rapidly bog down amidst the crossfire of the critical unresolved questions which we previously described.

4.1 Experiments in Building Prototype Sytems.

1. Studies of Existing Successful Support Systems.

We believe that a wide variety of useful insights can be gotten by close examination of the slowly growing number of successful support systems currently in use. Interlisp and EL-1, already mentioned, are two examples of successful development support systems which currently exist. The UNIX TM [4] operating system, along with its elaborate set of existing coordinated tools, seems to be a good basis for the construction of a variety of other support systems.

We believe it would be quite profitable to study these examples in an attempt to determine what makes them successful. It would be most useful, for example, to study their user interfaces to see which characteristics seem to make interfaces popular and useful. It would be helpful to determine, for example, what levels of HELP (tutorial) systems, graphics support, and directory assistance seem minimally necessary to insure utility and popularity in these systems. By studying a variety of support systems, we should furthermore be able to gain insight into which features are generally useful, and which are perhaps desirable only for limited classes of users.

Another important type of understanding, obtainable by such studies of examples, is an understanding of the importance of tight coupling of tools. It is maintained that the popularity of Interlisp and EL-1 derive directly and inherently from the specialization of their support tools to a single subject language. UNIX base support systems, on the other hand, are constructed from capabilities which, in general, themselves have no special language knowledge. Study of these examples can and must help us to come to an understanding of the extent to which language knowledge in tools is important. Study of examples and basic tool building research must then enable us to formulate notions of how language-intelligent tools and tool fragments can be efficiently created from a base of more general tools and techniques.

Study of these existing systems should also help us to better understand the data base/IMS requirements for environments. Each of these existing support systems seems to maintain to some degree of rigor and formality a central information base. The success of the various strategies in meeting user needs can surely be studied. In particular it would be important to study the type and amount of information retained, and the acceptability of these retainment policies to users.

Each support system has also adopted, apparently only implicitly, some model of its users and their activities. It would be interesting to formalize those models. From such formalizations could, for example, be determined the breadth of support extended by each system. User surveys could help determine the uniformity and strength of such support. This would help determine the ranges of applicability which are feasible for current support systems.

The models would also enable a determination of the flexibility currently offered by such systems. It has been hypothesized that environments must accommodate themselves to their users' way of getting their jobs done, not vice versa. It is important to determine whether current support systems do this. If not, it is important to determine whether this lack of flexibility is a significant source of dissatisfaction. It would also be important to decide if any such lack of flexibility is an essential consequence of tight integration, or whether perhaps better tool fragments could be used to achieve both tight integration and flexibility.

The list of things which should be studied in existing support systems could go on indefinitely. Perhaps it is best to close simply by observing that much experimental work can be carried out on currently existing support systems. This work should be used to guide the creation of new experimental systems in the direction of systems which can provide elaborative, rather than duplicative, insights and understanding.

2. Tool Fragment Studies.

This would be an experimental program aimed at identifying useful sets of tool fragments, and the extent to which they can support tight integration in the desired tool capabilities.

The experimentation would begin with the designation and assembly of a reasonable set of fragments to meet the needs of a specific, sharply circumscribed software activity. For example lexer, parser, flowgraph generator, verification condition generator, theorem prover, data flow analyzer, and test probe inserter fragments might be designated as the fragments necessary to provide total comfortable support for the verification and testing activity. An actual verification and testing procedure would then be hypothesized in formal detail. The selected tool fragments would be configured and integrated to support this activity. The experimental program would be continued by attempting to reconfigure the tool fragments in response to a variety of changes, such as changes in the testing and verification procedures to be supported, change in the source language, and change in the user community (e.g. the addition of managers as observers of the activity).

In the process of adapting to these changes, the tool fragments will be altered to provide needed flexibility, or perhaps the need for new or different sets of tool fragments will be recognized. In an important sense, this line of study is aimed at starting to understand what, if any, analogy exists in software production, to the manufacturing concepts of interchangeable parts and assembly line production. As in manufacturing, it is expected we will discover that the utility of these notions is not uniform. That is to say, we expect to find that useful sets of tool fragments can be successfully produced to form the basis for construction of some types of software environments, but perhaps not all. Thus experimentation with a wide variety of tool fragment sets seems indicated.

3. Data Base Studies.

This would be an experimental program aimed at determining ways in which informational bases for environments can be adequately stored, accessed and maintained. Following our conjecture that data bases for environments should be structured in accordance with models of the software jobs supported, this research project would begin by creating precise detailed models of various software jobs. This activity would be useful in itself as such models are extremely rare or nonexistent. From these models, data base schema would be designed and information management systems either built, adapted or simulated. The important experimentation would then entail the actual or simulated use of these data base /IMS's to determine the true performance requirements on them.

What must be determined are the sorts of demands which typical usage places on support system data bases and IMS's. Hence, for example, at first simulated streams of user requests should be directed towards the support system data base/IMS. (With the passage of time other research activities, such as the one described previously, should result in the creation of actual support system prototypes. These might be used to capture actual streams of user requests). Measurement probes, inserted in the IMS could then determine the searching, updating and deleting operations implied by these requests, and the required rates at which these operations must be performed. These measurements are needed in order to enable data base implementors to design schema which will facilitate efficient data base operations in support of expected user request sequences.

This sort of experimentation should help provide answers to the question of how much information the data base should store. Clearly information storage becomes excessive when it significantly hinders frequently occurring searching and/or updating operations. Actual measurements taken on a variety of simulated support systems should elucidate these issues. Presumably different types of support systems will be found to be amenable to the maintenance of different amounts of archival storage. This experimentation might also suggest useful hierarchical storage schemes.

From our discoveries about what constitutes excessively large data bases should flow a variety of useful derivative information. We should find out, for example, whether very broad scope environments do or do not place excessive demands on the data base and IMS's that must support them. We would find environments in which the rate of updating is sufficiently low to make the storage of redundant and derived information acceptable or even profitable. We should also be able to identify and perhaps characterize environments in which the storing of redundant information is impractical because of frequently occurring changes.

It is expected that this line of research might well serve as a stimulus for data base and IMS research, as we suspect that currently available technology will be found to be unsatisfactory in meeting the needs of environment data bases and IMS's.

4.2 General-Purpose Software Environments.

1. Construction of a General Purpose Environment to a Given Set of Specifications.

The purpose of this activity would be to actually construct a general-purpose software environment. It is assumed that this activity will not commence until after at least three years' experimental implementation, aimed at least partially, towards this goal. Hence this effort will build upon a base of successes in significant tool integration, in understanding the ingredients of a viable user interface, and in identifying software activities well enough understood and supported to be subjects for broad, strong, uniform support by tools. The totality of such experience should place active workers in the field in a position to clearly specify a general area of software activity and how it is to be supported by a software environment. An example of what seems achievable in this time frame might be an environment to support the needs of a medium sized team in at least coding and design of batch processing software written in a small set of closely related language dialects.

It should be stressed that this goal is not likely to be satisfactorily achieved without an experimental learning process such as was previously described. Starting from such a knowledge base, however, the goal of successfully producing environments of this scope seems reasonable. A program of continued experimentation and development should continue to broaden the scope of the environments produced, within bounds which should become more clearly understood as our ambitions grow.

2. An Experimental Test Bed for Configuring Environments

A far more ambitious and wide-ranging activity would be the assembling of a very wide assortment of tools and tool fragments with the goal of trying to configure them into a variety of environments. This activity would differ essentially from the previously described activity. It would not be directed towards the creation of single environments based upon the best products of earlier experimentation. Instead it would be directed at determining the range of environments producible from a fixed set of capabilities, and at evaluating the strengths and weaknesses of competing tools and tool fragments in the overall context of usage in a general environment. In a sense this is tantamount to a laboratory for environmental experimentation, and the activity supported, a forerunner of environmental engineering.

This sort of endeavor seems to entail greater risk, as we currently have a little experience in integrating software on so large a scale. Thus we currently have no basis for believing that such large sets of tools can be readily reconfigured to allow for the rapid construction of alternative environments which would be necessary for comparative evaluation. Perhaps the most realistic appraisal of this activity is that it is destined, at least within the next five years to be done by a free marketplace, rather than in a central facility.

References

[1] T.E. Bell, D.C. Bixler, and M.E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, SE-3, pp. 49-6Ø, (January 1977).

[2] "ECL Programmer's Manual," Center for Research in Computing Technology, Harvard University, TR 23-74, 1974.

[3] E.Ploedereder, "Symbolic Evaluation of User-Defined Procedures in EL1," Center for Research in Computing Technology, Harvard University, TR Ø1-79, 1979.

[4] D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System, Communications of the ACM, 17, pp. 365-375, (July 1974).

[5] S.A.Stephens and L.L. Tripp, "Requirements Expression and Verification Aid," Proceedings of the Third International Conference on Software Engineering, IEEE Cat. #78CH1317-7C.

[6] D. T. Ross and K.E. Schoman, Jr., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, SE-3, pp.6-15, (January 1977).

[7] W. Teitel, et al. Interlisp Reference Manual, Xerox Palo Alto Research Center, September 1978.

ADVANCED DEVELOPMENT SUPPORT SYSTEMS

Thomas Standish

Participants
Robert Balzer, Information Sciences Institute
David Barstow, Schlumberger-Doll
Meera Blattner, Lawrence Livermore Labs
Martha Branstad, National Bureau of Standards
John Buxton, University of Warwick
Adele Goldberg, Xerox Parc
Robert Morris, Bell Labs
Stephen Squires, National Security Agency
Thomas Standish, UC-Irvine

1.Ø Introduction

The purpose of the Final Report of the Working Group is threefold: (1) to provide an overview of the technical area addressed by the Working Group, (2) to summarize the group's discussions at the Workshop, and (3) to state the conclusions reached by the Group.

The specific technical area that our Working Group was asked to address was "Advanced Development Support Systems." We were asked to consider deliberately a long-range time frame. Specifically, we assumed throughout our discussion that the programming environments we were considering were intended for use a decade or more in the future.

Our discussions were wide-ranging, technically advanced, and very lively. They were catalyzed by a creative ferment that arose from the juxtaposition of the views of the practitioners and advanced thinkers.

During the first session of the Working Group, the participants examined two possibilities for the agenda that it would follow.

The first possibility was to decide on the characteristics of the user "setting" to be addressed by Advanced Development Support Systems and then to address a range of issue areas pertaining to that chosen user setting, such as software quality, the software lifecycle, software management support, program development tools, database properties, documentation, user interfaces, and maintenance. The rationale for adopting this approach was a recognition that programming environments are engineered artifacts built to suit given purposes and that variation in the purposes for which they are built can lead to variation in the technologies that are appropriate. Thus, it was viewed as important first to settle on the characteristics of the user setting with regard to such choices as single-user versus multiple users, production use or experimental use, long maintenance lifetimes versus short maintenance lifetimes, and so forth. Following the selection of an agreed-upon set of characteristics for the setting, it was thought to be profitable to examine the technical approaches appropriate to the given setting from a number of different vantage points, and then to attempt to synthesize the results.

The second possibility we examined for our agenda was to adopt a three-step "top-down" approach. The three steps are as follows: (1) Attempt to portray how software in the future might be different from software as we know it today; (2) Address the question of what software quality properties the programming environments we are building should incorporate; and (3) Attempt to characterize technical approaches, tools, and environment designs that will help produce software in the form identified in the answer to (1) and having the software qualities identified in the answer to (2). The rationale for this approach was that it could lead to a unifying, general view of the characteristics needed for general purpose Advanced Development Support Systems. If one believes, as did some of the participants, that software as we know it today will cease to exist, and that new forms of machine-based software will replace it, then it makes sense to adopt the "top-down" approach.

The Working Group decided to adopt the "top-down" approach instead of the "setting-driven" approach, and its deliberations followed the three steps outlined in the preceding paragraph.

The structure of this Final Report is influenced by the Group's choice of the "top-down" approach in the following sense. In the first three of the following sections of the Report, we cover the agenda items in the chronological order that they were discussed by our Group, namely: (1) What will software be like in the future?, (2) What is software quality?, and (3) What are the issue areas that need to be addressed in achieving software quality in Advanced Development Support Systems?

In addition to touching on a wide range of issues, the Working Group produced two views of future programming environments. These views are presented in the fourth section and serve as a basis for further analysis and discussion.

In the fifth section, we present research topics we feel need to be investigated in connection with the two views.

In the sixth section, a brief discussion is given of some of the individual reactions to the future views presented in the fourth section. When a first draft of this Final Report was circulated, the Working Group members commented in strikingly diverse ways, and this commentary, in effect, continued the work of the Working Group. The written reviews of the first draft thus became a source of material for the analysis and comparison given in the fifth section.

In the final section, we state what we believe to be some consensus conclusions achieved by the Group as a whole.

2.Ø What will software be like in the future?

If our vision of software is too much rooted in the present and insufficiently cognizant of likely future trends, we face the danger of presenting views of the future that will be technologically obsolete.

It is important, therefore, to attempt realistic predictions about how the characteristics of computer software might change in the early 199Øs in response to likely changes driven by two strong forces: (1) increased emphasis on software quality considerations, and (2) dramatically altered economic conditions for computing.

Another force that may play a large role in shaping the nature of future software is the emergence of practical, knowledge-based, intelligent systems to assist programmers and managers in the task of building computer systems. Though our approach to this is a bit more speculative than our approach to the likelihood of changed economic conditions, no vision of future programming environments can be judged to have been carefully considered in the absence of examination of the possible contributions of intelligent, mechanical programming assistance as an integral feature of future programming environments.

Regarding the influence of software quality considerations, we foresee increased emphasis in some important future settings on software quality assurance disciplines. Since improvement in software quality is a driving force behind the emergence of improved programming environments, we predict the emergence of better supporting tools and methodologies that are aimed at software quality assurance. It is likely that environments used in production settings will continue to adopt increasingly more powerful supporting methodologies and tools yielding improved engineering practice in response to the demand for improved software quality.

We predict improvements in software measurement techniques and in techniques for estimating software production costs and resource consumption. We foresee improvements, also, in management disciplines, communication disciplines (including documentation), and individual programming disciplines as the production engineering aspects of software science become better known, better validated, and more widely diffused in practice. We envisage that future programming environments will become highly-integrated and sophisticated in their response to these driving influences, and that special tools and techniques, such as the use of rapid prototyping methodologies will emerge.

Conservative estimates of the impact of the new VLSI technology indicate that in the not too distant future, we may have fast, cheap chip computers of substantial memory capacity sitting on our desktops. The need for communication and cooperative programming and the non-declining cost of peripherals will likely create the perception that it is sensible to arrange these desktop computers into networks that share expensive peripherals and common databases. Furthermore, the interconnection bandwidth in such local computer networks will likely be sufficient to support message communication and program and data transfer at very reasonable rates.

We view it as likely that such computing arrangements will improve substantially the computing power available to individual programmers and managers in comparison to that available in current time-sharing or batch processing environments, and our deliberations were conditioned by a consciousness that the nature of practical software might well shift, if the economics of computing were to shift, say, in the direction of making it practical to record substantial volumes of design decisions and design rationale, or to change the power and size of the resident, desktop software.

For example, if million-word, fifty-nanosecond desktop computers become available for on the order of a few thousand dollars, intelligent, automated programmer's assistants running on top of Interlisp with a half-million words of program and data will become feasible, whereas in today's time-sharing environments, they are often impractical. This shift in scale affects the consideration of what is both possible and practical in the early 1990s.

Thus, it is our view that the characteristics of software will shift in the decade of the 1980s in the following ways: (1) Software will tend less to exist in paper representations and more to exist inside the machine. Paper may tend to be used less and less as a container for documentation as the machine representations tie together various quality assurance practices and software communication disciplines for training, design validation, maintenance, incremental redesign, etc. (2) The availability of cheap bulk memory and the possibility that desktop computers will support resident software development tools of substantial size and at substantial computational speeds create incentives for software to evolve into forms in which designs and refinements are captured in the machine and are subject to extensive annotation, indexing, and commenting. It may be possible to capture decisions, rationale, refinement transformations, version derivation records, and the like, and to derive parts of the software by means of tools of substantially increased automatic (or human-assisted) power. (3) Incentives will exist to drive software into a form that resides in a database and provides hooks and handles for a high degree of integration between the tools and management disciplines used in its development. Software products may become less distinguishable from the environments in which they were developed. It is possible (likely?) that programs as they are conceived today may cease to exist.

3.Ø What is software quality?

Inasmuch as a principal aim of the evolution of future programming environments is to assure the development, maintenance, and upgrade of quality software, it is important to inquire into the nature of software quality. What is software quality?

In our view, software quality is multi-dimensional. Nearly every software system exhibits a number of general dimensions of quality such as: (1) reliability and correctness, (2) efficiency, (3) maintainability, (4) transferability, (5) responsiveness to user needs, (6) timeliness of delivery, and (7) unit cost.

In addition to these general software quality properties, particular software systems might exhibit special dimensions of software quality related to their context of production and use. For example, some software systems might have quality goals relating to: (1) meeting real-time constraints, (2) fault-tolerance, (3) self-diagnosis, (4) modifiability in the face of changing requirements, (5) commercial marketability, and so forth.

Pursuit of software quality along these various dimensions may involve trade-offs. For instance, increasing efficiency may trade-off against low unit cost.

On the other hand, pursuit of some goals may simultaneously help in the pursuit of others. As one example, enhancing maintainability during pre-release development may lower overall lifecycle costs. As a second example, production of software that is initially efficient, correct, and responsive to its requirements increases chances that it will be delivered on time.

In general, in considering software quality goals, no single goal is to be pursued at the expense of the others and no single optimization criterion can be established. There is no point in maximizing one dimension of quality at great expense when so doing reduces some other dimension of quality below a required threshold level.

Thus, we might profitably view software quality goals as simultaneous constraints to be satisfied rather than as measures to optimize independently of one another. From this viewpoint, if programming environments are built to help achieve software quality goals, then programming environments can be viewed as constraint satisfaction systems.

Some software quality goals may be viewed as secondary goals rather than as primary goals, in that their achievement indirectly supports the achievement of primary software quality goals. For instance, if we look at the goal of "simplicity" (in the sense of trying to achieve the least complex design that meets given requirements), we see that "simplicity" is not an end in itself. Rather, it is a goal, which, if achieved, supports other software quality goals such as implementability, low risk, maintainability, and ease of learning.

Thus, when we attempt to justify why a given programming environment has certain features, incorporates certain tools, or supports particular disciplines, we may construct justifications either by direct appeal to primary software quality goals, or by reasoning chains that show how primary goals are attained indirectly by achieving one or more intermediate goals.

There are also software quality issues that arise in connection with the software lifecycle. Because we live in an imperfect world, where requirements are never likely to be complete or accurate, designs are never likely to reflect the requirements perfectly, and implementations are never likely to satisfy the requirements or to reflect the design intentions perfectly, we must resort to special measures and disciplines in order to improve software quality progressively.

Thus, testing, on the one hand, and maintenance, on the other hand, occupy prominent roles in the software lifecycle because of imperfection in earlier lifecycle stages. As another example, the idea of rapid prototyping methodologies makes sense because they are a means whereby we can construct working initial subsystems that create the functionality that the user sees, so that we may detect imperfections in our understanding of the requirements and so that we may accelerate the learning process by which we discover and articulate the true user needs. From the point of view of constraint satisfaction systems, rapid prototyping allows early satisfaction of some of the constraints.

At a deeper level, we see that there are feedback loops between the activities in the software lifecycle that help us incrementally to improve the understanding and the software quality achieved at each stage. Thus, we may only really begin to understand the requirements when we are exposed to the behavior of an implementation. Cyclical exposure to the behavior of the artifacts we build may be necessary to achieve understanding of the true requirements, especially for a system we are attempting to synthesize for the first time.

We also see that a number of software management practices sprout from the necessity to utilize reliable methods for achieving software quality, wherein we attempt to detect and remedy flaws in the quality of visible products in each of the lifecycle phases as they participate in feedback learning and improvement loops.

Seen from the software quality viewpoint, adequate production programming environments for programming in the large must support management disciplines to schedule activities, configure resources, engage in quality assurance monitoring, and adjust incremental effort to achieve the required quality in each of the visible lifecycle stage products.

The nature of the tools appropriate for programming environments is critically affected by these processes of feedback learning and incremental improvement. For example, if we anticipate that requirements statements and design statements will change in response to feedback learning from later lifecycle discoveries, then it profits us to consider the desirability of trying to capture requirements and design decisions in machine-manipulable form, and to have tools available with which to keep them up to date as a project evolves.

In summary, because each lifecycle stage takes place in the context of imperfect predecessors, we are led to consider feedback loops which lead to incremental improvement of the products at earlier stages. These interplays have impacts both on the tools we need in programming environments and on management disciplines needed to manage activities at each of the lifecycle stages.

Many serious problems would disappear if we lived in a perfect world where requirements were perfectly and completely articulated, where designs were consistent, complete, and perfectly reflected the intentions of the requirements, and where implementations perfectly realized the designs. For instance, the need for testing, design walkthroughs, independent validation, and so forth, would entirely disappear. But part of the complexity in building effective production programming environments comes from dealing with imperfect and incomplete information as "boundary conditions" for each of the lifecycle activities.

Another important topic that comes up in connection with software quality is that of software "metrics". How do we measure what has been accomplished so far on a software project, and how can we accurately predict what remains to be accomplished? How can we measure software properties such as efficiency, unit cost, size, partial correctness, degree of completion, and so forth? We know that simple bug counts are not a reliable indicator of how many bugs remain to be discovered. We know that simple counts of the number of lines coded is not a measure of the degree of project completion.

With regard to the art of estimation, we need theories of how the dependent variables we are trying to predict depend on independent variables we can measure. For example, how does degree of project completion depend on measurements of partial correctness and on lines of code written versus lines estimated to be needed, if at all? While the science of "software measurement and prediction" might still be regarded as being in its infancy in terms of our known stock of validated theories of practical utility, it is clear that a mature science of "software metrics" would substantially improve our chances of building production programming environments in which we could produce quality software that is reliable, efficient, correct, responsive to its user needs, and is delivered on time and within budget.

Another set of issues that affect our considerations of software quality are those involving relationships between differing kinds of expertise required to create and maintain software systems. Often, in practice, we may need to congregate teams of specialists with expertise in different areas when building, debugging, or upgrading software systems, and it may be necessary for these teams of experts to cooperate.

For example, in building an inertial navigation module, we may need to draw on the knowledge of a physicist to design or upgrade portions of the system relying on knowledge of kinematics, dynamics, coordinate systems, and the like. We may need to draw on the expertise of a numerical analyst to design or debug portions of the system connected with accuracy decay in updating a description of current position derived from accumulating piecewise incremental measurements. And we may need to draw on the knowledge of expert programmers to implement or debug the system in executable form and to settle questions about nomenclature scoping rules for local variables in procedures, and the like.

It may not be possible for any of these three types of experts to answer questions or to reason effectively in the specialties that are the trained province of the others. Not only must such teams of experts congregate and cooperate at system design time, it may also be the case that we must recongregate such expertise if we are to debug or upgrade a system during the software maintenance phase of the lifecycle.

In considering the design and implementation of the system, problem solutions expressed in languages familiar to each of the experts who must contribute to the system design at different levels must be translated into linguistic expressions at lower representational levels. That is, we must be able to shape the data and operations available at each level of representation of the system so as to imitate the behaviors required at the next level up in order ultimately to realize the behaviors required at the user level.

Thus, there are software quality considerations related to the accuracy and adequacy of representational processes at the various levels of linguistic abstraction spanning the gap from the naked machine up to the application domain, and there are structural and economic questions involved in training new personnel to understand how multi-layered systems work and in bringing to bear the right kind of expertise and capability to diagnose, repair, and upgrade the system during maintenance. Effective programming environment tools, including documentation and training tools, must address these issues adequately.

4.Ø Some issues we discussed.

In this section, we summarize some of the issues that were discussed during the Working Group's deliberations subsequent to our examination of the nature of future software and the nature of software quality. These issue summaries are not given in the strict chronological order we treated the issues, as the discussion often provided incremental contributions to our understanding of certain topics in an interleaved fashion. Thus, the remarks are collated and sorted by topic from notes taken during the meeting.

Noncumulative Science?.

A troublesome issue that recurred throughout our discussions was whether or not we were engaged in a cumulative science. That is, have we been able to arrange matters so that we stand on each other's shoulders in order to make cumulative progress, as opposed to stepping on each other's toes. In general the participants agreed that, in the present state of software science, we do not have very effective mechanisms for finding out what history says about a particular problem or what solutions history suggests. This concern reflected itself in a positive way in our conclusions in terms of an agreed-upon desire to identify technical approaches to effective reusability of software. It also led us to wonder how much of our own deliberations consisted of a replay of history unknown to us.

Documentation.

Some of the participants advanced the view that documentation should be a byproduct of the development process. This was envisagd to consist of saving the transformations, design decisions, and rationale used in system development. One participant took the position that there exist structures of decisions (such as "divide-and-conquer") and that to understand a program, someone must understand its underlying structures of decisions. One observation about the prospects of mechanization of such decision capture was that we should expect transformation sets always to be incomplete and open-ended and that provision must be made for extension.

One view of documentation capture is that the programmer (or designer) should announce what he is doing while he is implementing a system. But this leads to the problem of "excluded magic" in which the expert can't say what he is doing since he is unconscious of it. It thus becomes an important technical problem to be able to identify those points at which the programmer may be unconscious of the knowledge he uses. Can a machine be programmed to identify those points where there is a fact about the domain or a belief about the domain that needs to be exposed?

Management Support Capabilities.

Whether to discuss the topic of management support capabilities was a source of controversy to the participants. One participant felt that management support capabilities were of more immediate concern than of long-range concern, and that it was therefore not in the charter of our Working Group to discuss them. (Our charter was specifically to address long-range issues.) Another participant felt sufficiently strongly about the irrelevance of management support that he absented himself from the deliberations until the discussion of management support was concluded.

One participant with a background as a practitioner noted that there are different management styles in different organizations and that programming environments might profitably adopt a "tool kit" approach that allows an organization to assemble a management discipline tailored to its own particular philosophy, standard operating procedures, and the quality assurance practices.

A number of participants justified the necessity for attention to management considerations as follows. In any given state of development of software science, no matter how powerful the tools available to a single user, it is likely that somebody will want to attempt construction of a system requiring an order of magnitude more power than that available to the single user. That is, ambitions may always outpace the power available to a single user. Under such circumstances, we will need to combine the power of several users and they will need to cooperate, collaborate, and communicate. It was further emphasized that lots of software development in the U.S. Government is not single user development, and that the Working Group would not adequately adhere to its charter if it failed to address multi-user activities. Whenever there is a need for communication, collaboration, and apportionment of effort, there is a need for management, and so, in at least some settings of relevance to the Workshop sponsors, management support cannot be ignored.

Settings.

It became evident during the course of the discussions that different participants had differing ideas in the back of their minds about the characteristics of the settings to be addressed by programming environments. Some were interested in providing maximum leverage to the single user to amplify his programming power. To others, security was irrelevant since they had no need to work in secure environments. To others, concern for increased power for the individual programmer was of less concern than increased power in quality assurance practices for teams of people since that is where they perceived there to be critical leverage.

Thus, by choosing the "top-down" approach, the Working Group tacitly failed to identify the characteristics of the settings that environments were to address. This led to frank disagreements about the importance of various issues and concerns with respect to the tacit assumptions about settings each participant held but was never asked to state explicitly.

Languages.

One participant identified four possibilities for languages appropriate to programming environments: (1) a new language is devised for each application domain, (2) a wide-spectrum language is used which contains many special sub-languages all present at once, (3) there is a tower of refinement levels of language moving from concrete implementation languages through modelling languages up to domain specific languages, and (4) one uses different distinct non-interacting languages for design, programming, and requirements analysis (e.g., SADT, flowcharts, and COBOL).

Some thought that the appropriate user language is the language of his domain of application. Others thought that the system specification language should be operational in the sense that one can execute it and analyze it. One participant held that if one uniformly uses an "evaluable specification" the methodology can be applied across the range of languages from the application domain to the implementaton domain. Another objected that such an approach may take away from the application domain user some kinds of concepts he is interested in using, and that in some situations, procedural languages may be unusable and declarative languages would be needed. A practitioner-oriented participant stressed the importance of developing better understanding of the multiplicity of domain specific languages and how they were related to the software lifecycle.

Granularity and Composability.

One participant felt that having a small grain size in the environment tools and in the units of interaction with the user enabled one to get surprising, useful compositions never dreamed of in advance. Another observed that software reusability may be inversely proportional to grain size --- i.e., the smaller the grain size of a tool, the more likely it is to be reusable. For example, in future environments, there may be no single tool identifiable as a "compiler". Rather compilation may result from composing the actions of smaller size tools such as parsers, program improvement transformers, and code-generators for one or more target machines.

Training.

Many participants felt it was a major research topic to investigate how to train programmers to understand the design, construction, and maintenance of a system. Some felt it crucial to have excellent training in the use of future environments as a prerequisite to their eventual technology transfer and deployment.

Security.

An issue of concern to some was that of security since, as a practical matter, they felt that a number of important user settings, characteristic of government and industrial contractors, were required to address security. Security was viewed as achievable in layers that raise the economic cost of penetration by perceived threats. That is, there exist techniques that make it expensive to penetrate a system and that lower the bandwidth of the penetration channel. In this context, one must balance risk against threats at given levels of economic investment.

It was observed that implementing a security policy in industrial settings is costly and inconvenient, so that anything programming environments could do to provide for, e.g., trusted operating systems, secure data, isolable systems, and so on, would greatly enhance the attractiveness of environments in certain settings, whereas failure to address such issues might mean that environments were unusable in practice despite other advantageous characteristics.

Software Measurement and Prediction.

Several of the participants noted that the state of the science of software measurement and prediction is currently unsatisfactory, and they emphasized that improvements in this area would be welcome as an integral part of effective environment technology.

For instance, it is known that measuring the amount of code written is not sufficient as a progress measure and that bug estimates are insufficient for assuring software quality. Yet there is a clear need for metrics to measure progress toward software project goals.

5.Ø Two Views of the Future

In this section, we present without analysis or commentary, two views of future environments that were created during one of the days of the NBS Workshop by members of our Working Group.

Because the time for their creation and refinement was short, the views are brief. Nevertheless, their subsequent examination led to stimulating discussion by the Working Group about appropriate research topics to investigate, and about what technical approaches may have promise. Thus, they played an integral part in our evolving discussions and analysis, and they helped us to reach our conclusions.

THE BUXTON VIEW.

> "A software support system is intended to provide long-term support for large, long-lived and evolving software projects. The support includes programming-in-the-small; that is, designing and constructing modules in a programming language as system components supported by suitable design and implementation tools. Even more critically, it supports programming-in-the-large; that is, overall system specification and design and building and controlling over long life-times the different configurations and versions of the project which arise in response to the evolution of requirements. This is a less well understood area and requires new and different languages and tools; for example, declarative rather than procedural system description languages.
>
> A basic ingredient of the methodology of such a system is the realization that a software project, in the present state of our knowledge of the subject of software

engineering, is best developed through a series of prototype stages. The support system must provide tools to support the generation and testing of a sequence of prototypes. It must provide the ability to generate production versions of the project from selected members of the sequence which are crystallized out to provide service to end-users of the project.

The development of such a system, which is clearly a large software project, is itself best undertaken as a series of prototypes. In the later stages of its evolution such a system is itself one of the projects which it supports; that is, it becomes self-supporting. Furthermore, it enables some benefits to be transmitted from previous experience, acquired both in its own development and in the develoment of the projects it supports.

THE BALZER/BARSTOW/GOLDBERG VIEW.

The Goal

The Goal is to specify a system development environment in which an operational model in the language of the domain can be used as a prototype of the intended system and can be refined into an efficient implementation.

Users

The Specifier is expected to be an expert in creating models of the pertinent aspects of his domain.

The Refiner is expected to be an expert in realizing models as efficient implementations. In the long term, portions of this role may be automated.

Philosophy

An operational prototype forms the interface between the Specifier and the Refiner, and separates their activities.

The Specifier proposes a model of the pertinent aspects of his domain and uses the operational characteristics of the model to determine whether it matches his intent. Typically, the first model will be functionally inadequate and the Specifier will iterate this modelling activity until an acceptable operational prototype is obtained.

Upon completion of the above specification activity, the Refiner, through a process of multi-level design, realizes this operational prototype as an efficient implementation.

This design process is recorded as documentation and as the basis for later redevelopment (maintenance). The resulting implementation is never modified; rather, the design process is modified to yield a new implementation. The feasibility of this maintenance approach rests upon the reliability and the economics of the redevelopment process.

The system development environment should eventually be capable of supporting itself.

Essential Components
(all machinable)

1. Mappings Between Levels of Concepts

Design concepts are organized in a multi-level fashion. Each level consists of an operational or at least analyzable specification.

There exist mappings which can be applied to a concept or concepts at one level in order to convert them to concepts at other levels. The validity of the resulting implementation depends on the validity of the mappings.

2. Languages

There are a number of languages required, such as:

Specification	Language
Refinement	Language
Mapping	Language
Documentation	Language

3. Viewing Behaviors

The Specifier must be able to determine whether or not a proposed prototype (even an incomplete one) matches the system's functional requirements. This implies that the prototype is either operational or analyzable and that the resulting behavior can be observed in understandable form in the language of the domain.

4. Documentation

The refinement activity is captured as a record of the design. This record can be reformulated to provide a clear top-down presentation of the development. This reformulation and record provide a readable (understandable) basis for future redevelopments. As such, it must include the justification supporting design decisions as well as the decisions themselves.

5. Repository

Two information sources are available:

(a) An encyclopedia of the cumulative results from Computer Science used to map concepts from one development to another. These mappings are chosen mainly on efficiency criteria.

(b) A library of previous designs (not merely resulting realizations). These designs can be redeveloped (as in any other maintenance activity) to assimilate the functional unit into the current activity. This provides a general reusability capability. Such modifiability becomes more critical as the size of functional units increases.

Usage Scenario

1. Specify the System

1.1 Use the operational aspects of the specification as a guide for further elaboration of the specification as a prototype for determining the system's functional behavior.

1.2 Compare operational behavior with desired Functionality. This comparison relies on the ability of the system to produce readable and understandable descriptions of the prototype's functional behavior.

1.3 In case of mismatch, modify system specification and repeat behavior comparison. This step is a validation of the proposed model at the domain concept level. This rapid prototyping capability allows significant aspects of the system testing to occur before implementation. This permits the Specifier activity to be independent of Computer Science expertise.

1.4 In the event that the domain specification language is not fully operational then some refinement will be needed in order to reach an operational prototype.

2. Map specification into implementation

2.1 This step is needed only if the operational specification is not sufficiently efficient to be used as an implementation (or fails to meet some other implementation criterion).

2.2 This mapping process is driven mainly by efficiency concerns and is dependent primarily on Computer Science expertise. However, the applicability and/or advisability of any mapping may also be dependent on domain properties.

2.3 At each step in this design process, the Refiner has the option of either:
(a) selecting a mapping from the encyclopedia, or
(b) assimilating a previous design from the library by redeveloping the design, or
(c) expanding the set of mappings or designs in the Repository

2.4 The validity of the resulting implementation derives from the process of its development, through the validity of the individual mappings and designs employed. Testing is still required because these mappings may be incorrect.

2.5 The attempt to apply a mapping may uncover incompleteness(es) in the specification, e.g., insufficient constraints. This requires an elaboration of the specification and a rederivation of the design.

2.6 A structured presentation of the design is needed as an enhancement of the recorded design to facilitate understandability of the process during future developments.

2.7 The operational nature of the developing implementation provides an opportunity to highlight remaining critical decisions by instrumenting or analyzing the current (incomplete) design.

3. Maintain the system

3.1 Maintenance is the process of assimilating the previous system design. Redevelopment of the previous system design is required as a result of either:
(a) enhancement or upgrading of the specification,
(b) tuning of the design,
(c) new implementation criteria, or
(d) errors in the mappings employed.

3.2 This maintenance activity is an instance of refinement, Step 2.3. As such, the expertise required by a Maintainer is the same as that of a Refiner.

3.3 Enhancements (upgrades) of the specification are made by the Specifier in response to changed user requirements.

3.4 In response to expansion of the Repository, the system may issue notifications of possible opportunities to assimilate the expansion into existing designs.

6.Ø Research Topics

In order for programming environments based on the views given in the previous section to be realizable as effective working production systems, a number of research topics need to be investigated.

We have identified the following (more or less urgent) research topics that must be investigated if the Buxton View is to be satisfactorily realizable in production versions.

Short-Term Urgent Research Topics.

1. Rapid Prototyping Systems, particularly those that are rapidly responsive to evolving requirements.

2. Management Capabilities for programming in the large, particularly managing huge configurations and production of system variants.

3. Sophisticated Automated Test Methodologies, including automatic test case path analysis, test case coverage analysis, and test case generation.

4. Databases for programming environments, particularly history-taking and derivation management.

5. Understanding Software System Levels, particularly doing abstraction correctly and making progress on techniques for domain specific modelling languages.

Since the Buxton View relies heavily on progressive improvement of prototypes, careful version and configuration control, and on effective testing methodologies to assure software quality, it is clear that if the above research topics can be satisfactorily investigated and if good results can be attained and transferred into practice, the Buxton View could be realized in a production version, say in the early 199Øs.

The construction of a practical, working, production programming environment along the lines of the Balzer-Barstow-Goldberg View relies on the attainment of a number of long-range research goals. For example, if an automated on-line Encyclopedia of Computer Science knowledge is to be built, we must understand how to codify programming knowledge in machine usable form. While important steps have already been taken in this direction, and while the results are by no means meager, it is clear that more remains to be done before the Encyclopedia of Computer Science expertise can be automated and applied to the mapping of programs in problem specific language into efficient underlying implementations without substantial human intervention at the detailed transformational level.

In the following list of long-range research topics, we have interwoven topics that are essential to investigate before the Balzer-Barstow-Goldberg View can be realized along with other topics of substantial interest to us, whose investigation would have strong benefits for understanding how powerful programming environments could be built.

Long-Range Research Topics.

1. Modelling the design process, including: (a) theories of multi-level design and mappings across conceptual levels, and (b) the nature of maintenance and incremental redesign.

2. Codification of programming knowledge, including that which is (a) human usable, and (b) machine usable. Research issues here include: (a) What is it?, (b) How do you use it?, and (c) How to structure it.

3. User interfaces, including both physical interfaces and languages, and identifying different points of view with which users can approach the system.

4. Specification languages, including the spectrum from: (a) domain independent specification languages to (b) domain dependent languages.

5. Modelling the user of the environment (either the Specifier or the Refiner).

6. Prototype environment developments

7. Management models

8. Understanding the educational process, including identifying and validating effective means of training users in how to use advanced environments.

9. Evaluation and analysis of high level models, including those that are: (a) declarative, and (b) procedural.

10. Understanding technologies for distributed knowledge sources, including shared use, update, and coordination.

11. Developing a good cognitive or psychological theory of system understanding. Some issues are: (a) What is a good explanation of a system or program?, and (b) What makes system explanations readable or comprehensible? Some possible techniques to investigate are: (a) use an initial, simple, incorrect model and use exceptions to improve it progressively into an explanation that is complete and accurate, and (b) present simplified views in intellectually digestible chunks that isolate and study subsystems of a system, and then

consider interactions between the views.

12. Browsing. What are good techniques for browsing a large knowledge base to identify knowledge relevant to some set of needs?

7.Ø Brief Analysis and Comparison

In this section we briefly analyze and compare just a few aspects of the two views of future environments given in the previous section. A number of the observations in this section were received as written reactions to an earlier draft of this Final Report.

With regard to the Buxton View, we need to spell out what we mean by a prototyping methodology in greater detail. We need to identify perhaps several alternative technical approaches to prototyping, we need to understand how the behavior of prototypes is supposed to be reviewed by system users, and we need to know how that review can lead to systematic examination and update of the system requirements statements. In this connection, we need to be concerned with what form the representation of the requirements will take, and with the overall methodology for incremental improvement of the requirements statement that is supposed to result from the use of prototyping methodologies.

Another observation about the Buxton View is that it diverges from the Balzer-Barstow-Goldberg View in that the Encyclopedia of Computer Science Expertise, which is integral to the design of the latter view, does not derive from the Buxton View. That is, one has to codify and express computer science expertise as an activity separate from those envisaged in the Buxton View.

With regard to the Balzer-Barstow-Goldberg View, some participants felt strongly that there should be additions made to assure that it would be useful for programming in the large. These participants felt it would be a useful step to indicate additional steps to be taken to accommodate programming in the large, and that while there was insufficient time in the context of the Workshop to give this subject adequate thought, nonetheless the subject should be acknowledged in the Final Report. One participant noted that the Balzer-Barstow-Goldberg is not adequate to the mission of the National Bureau of Standards until it addresses programming in the large.

Some participants expressed skepticism that the research goals needed to make the Balzer-Barstow-Goldberg View viable could be attained in a reasonable period of time. One participant commented as follows on this subject: "Being 'far out' is not a sufficient goal. No matter how far out some notion might be, it does not really do us much good unless we know how to reach it from some foundation. This is fundamental to the notion of computing." The attainability of the Balzer-Barstow-Goldberg View might be better understood if it were possible to give a reasonable estimate of how much codified computer science expertise might be needed and what scale of effort might be needed to produce it.

8.Ø Conclusions

Some conclusions that were more or less agreed upon by the participants are as follows:

1. Rapid Prototyping --- A development system is required that facilitates quick, inexpensive creation of operational prototypes or models. Solutions to many software problems are not achieved in a straightline fashion, but rather through progressive refinement of a model starting with an initial trial solution. The intermediate solutions are progressively improved many times until a satisfactory solution is achieved. Systems that facilitate this solution refinement and that make it affordable to solve significant problems more than once are needed.

2. Layers of Refinement --- A progressive refinement from application domain model to machine implementation was viewed as a basic mechanism for creation and representation of the product. As technology advances, the refinement from one layer to another should become increasingly more automated. Currently the mappings are done manually. Eventually they should be incorporated within the system itself. In either case, the mappings or transformations should be recorded as part of the system documentation.

3. Documentation --- Documentation should be viewed as a by-product of the production process. Recording or saving the product alone is insufficient. The production process must also be saved so that it can later be modified to produce a new product which is a variant of the old. Thus, not only the final product, but the intermediate stages and the mappings between them must be part of the documentation. The capture of this information should be performed by the system with little or no intervention from the developer.

4. Maintenance --- Maintenance is an activity encompassing both error removal and system upgrade. The system upgrade portion of maintenance should be viewed as incremental redevelopment. Systems should be constructed so this view is economically viable. Products should never be modified by changing a piece of code. Instead, the product should be redeveloped, as required, so that requirements, design, and

implementation are always coherent refinements of one another.

Underlying the very concept of a development environment is a need for better organization and management of information. Advances in this area affect the ability to achieve the documentation and maintenance approaches described in the previous paragraphs. Considerable amounts of information concerning the process of program creation must be stored and organized to be accessed later from various and changing points of view. Information must be managed to support cumulative aspects of the science. Programs and the development process must become reusable.

HIGH LEVEL LANGUAGE PROGRAMMING ENVIRONMENTS

Marvin Zelkowitz

Participants
W. Richards Adrion, National Science Foundation
Alfred Aho, Bell Laboratories
Daniel Bobrow, Xerox Corporation
Thomas Cheatham, Harvard University
John Cherniavsky, National Science Foundation
Susan Gerhart, Information Sciences Institute
Gordon Lyon, National Bureau of Standards
John Nestor, Carnegie-Mellon University
Terry Straeter, General Dynamics Corporation
Marvin Zelkowitz, University of Maryland

1.Ø Introduction

This report is a summary of the group on Requirements for a High Level Language Programming Environment that was part of the National Bureau of Standards' Workshop on Programming Environments that was held at Rancho Sante Fe from April 29 - May 2, 198Ø. Although written by a single author, this report incorporates the views of the entire group that spent long hours in early May working towards a consensus view of programming environments. The author wishes to thank all of them for their help.

2.Ø The Model

Software is often delivered late, is unreliable, is not maintainable and has a host of other well known problems. In order to look at possible improvements, a model programming environment was developed. It was believed that this model could incorporate most of the ideas needed to greatly improve programmer productivity. This model is sufficiently general so it is usable even if some of the needed research results, mentioned later, are not fully developed. Although the use of high level languages is rarely explicitly mentioned, the underlying assumption is that such a set of languages is needed to interact with the implementation of the model. More about this later.

Assumed hardware and software technology circa 199Ø was the starting point of our discussions. Although mentioned as a date, it was used only as some unspecified future milestone - we explicitly did not want to be bound by any near term (within next five years) developments since these would be adequately covered by two of the other three groups at the workshop. We assumed that any such system that we proposed would probably be built from scratch, and would not need to have its goals compromised by the reality of extending an existing system. Experience of several of the members in the group on current day experiments in programming environments (PIE [2], PDS [1]) gave us first hand knowledge of some of the problems in developing such systems.

The basic model consists of a set of objects that are hierarchically related. An object is a collection of entities, where an entity is a set of (attribute, value) pairs. Attributes are things like author, source code, specifications, date of creation, and version number. By having a two level structure of attributes and entities, the same concepts can be viewed in different ways by different individuals. Thus each user of such a system would have his own perspective of the environment, i. e., each user would have his own view environment as part of the larger programming environment.

For example, a given module might have the attributes of author, source code, specifications, cost, and schedule. The programmer's environment would contain source code, schedule, and specifications, but might not contain the detailed cost. A project manager would need the schedule, cost, and probably the specifications, but would need the source code. There is only one copy of the actual source program or specifications, and numerous objects are able to reference these indirectly via its entities.

The basic operation of this system is a set of transformations that map objects into other objects. A compiler transforms a source program object into a relocatable object in a program library. A verifier checks whether the source code object implements the specifications object. Editors, optimizers, verifiers, compilers, and other software tools are examples of transformers that can be built using today's technology.

However, the real power of such a system is in applying new transformations that can be added to such a system. Applying a transformer that takes a machine description (in some language like ISP, for example) and an abstract algorithm into a specific source code object should aid in portability and in maintaining multiple versions of an existing system. Changes to the specifications of a running system can be propagated more easily into the source code if many of these transformations are automated, and thus help in the maintenance phase. These and other research objectives will be described later in this report.

Objects are hierarchically related and can be interrogated at any level. At the top level would normally be the abstract requirements and the bottom level would normally be the concrete code to execute the intended function. The system can be interrogated at any of these levels, and the structure derived from these interrogations. Thus it would be possible to trace back a given source code function to one of the requirements. Similarly, changes in requirements should automatically reflect themselves as changes to the source program.

The environment will keep history information about all objects (i. e., where they came from). Such a structure would greatly aid in managing a large software development. Configuration control would be possible and maintenance, the major cost-factor in current day developments, should be more under control since all changes should be traceable. The use of transformations should aid in rapid prototyping so that models of a given programming project can be tested before great expenditures of time and money.

Every object has a contract attribute. This concept generalizes the ideas of program interfaces and assertions. At the highest level the contract gives the requirements that the object has to meet, while at the lowest level it would represent the formal specification of the function that the source code implements. A program verifier could check the consistency of these. In addition, interfaces between two modules can be checked in a more thorough manner. This is an extension of the simple data input-output interfaces checked by PSL/PSA or the package structure in the language Ada. The term "contract" is used intentionally instead of the more common "assertion" or "formal specification" in order to get away from the notion of verifying only source level statements.

The system is also self-descriptive and contains facilities for measuring its own performance. Any user should be able to interrogate the system to determine the status of the environment and what is expected next (e. g., HELP files on many current-day systems). This allows novices (and experts, too) to manipulate objects more easily. The measuring characteristics of the model will enable management (of the environment itself or of some software project being developed in the environment) to control operations better.

This model incorporates most of the ideas that have been proposed in the last few years. However, what is most attractive about it is that it is flexible enough to be built without most of the above mentioned features. Given the appropriate underlying data base, the basic object relations can be built today. It would be easy to incorporate an environment, such as described by the Department of Defense Stoneman Document for the programming language Ada, within this structure [7]. In a later section of this report the various research topics needed to achieve an efficient implementation of this model will be described. Even if some of these research ideas are not fulfilled in the near future, the remaining ones can be incorporated in a practical production programming system.

Experience with unit development folders shows that there is some real hope that this model might be quite productive [4]. Unit development folders are an effective management tool in use today, and have proven to be effective in organizing data about individual components in a large developing system. Such folders contain all of the data pertinent to such a component. The model just developed can be considered as an abstraction of such a folder, thus it gives credibility that this model may in fact be quite usable in a production environment.

3.Ø The Setting

Given the basic model, the major question is how to build such a system. An important related question that then arises is for whom is the system to be built? It was decided that it should be built for experts and not for the large mass of programmers. By experts we mean both those knowledgeable in computer science and those experienced in the application area that needs the new software. We intentionally ignored the less sophisticated user since we realized that since we cannot yet build a truly effective environment for experts, we have no hope of helping everyone. If we had such an environment, then we could tailor it to others by restricting the set of transformations that can be applied.

We have also ignored as a major requirement the single programmer working in a laboratory (although he should be helped by whatever develops), since this is not where the current problems in software productivity arise. Thus we are considering the multiuser environment where individuals need to communicate with each other. Projects are generally multiyear, with a changing staff. The use of interactive terminals (as opposed to batch) is assumed, as well as a sufficiently powerful computer system to give each user a large amount of computing power (more about this later).

In exploring this model, however, we must be careful to avoid several pitfalls. We must not build a system only for ourselves (basically expert computer science researchers in a specialized laboratory setting). The resulting system must be useable in a large industrial setting (with an appropriately trained staff) and it must be able to be used to develop large software packages by large groups of programmers. It must also take into account real-world constraints like staff turnover, deadlines, long system lifetimes, generally lower level of expertise by the staff and other issues. We believe that this report addresses these issues.

One minority viewpoint must be mentioned here since we believe that it is unlikely, but possible, to happen. The view is that only single person projects need be considered since an individual's productivity would increase by so much that most projects currently taking large groups could be done by individuals working alone. This may very well be true; however, the majority view was that after considering the changes in programming over the last 30 years, we would simply increase the complexity of the tasks we set out to program - thus keeping the multiprogrammer environment active for many years.

4.0 Goals of Model

Before describing the details of the model and the research needed to develop such an environment properly, it must first be answered why such a model is needed, and how it differs from existing systems. Most problems in software today occur as communications problems. Requirements are poorly understood and are translated into ambiguous specifications and designs. Programmers have inadequate information about the interfaces among various modules in a system. Maintenance is difficult since different sites have different versions of a system and it is difficult to perform configuration control.

However, many of today's tools are oriented towards correcting source level programs. But actual coding only takes about 20 per cent of the effort on large programming projects. Thus most of the future benefits in improved productivity and reliability will be a result of improved communications and interfaces.

The proposed model of section 2 is based upon strict communication paths between parts of a system (called objects). In addition the model is sufficiently general so that almost any superstructure or methodology that adheres to this communication protocol can be built on top of it. We believe that this will be most effective in increasing information transfer among the components of a project.

5.0 Research Topics

In order to realize the model described in section 2, the current field of computer science was surveyed and the known problems needed to be solved were identified. These were broken down into 5 basic areas:

1. Environment Issues. The description of objects and the various transformation tools needed for them.

2. Knowledge and reasoning issues. The basic ways the environment helps the programmer (e. g. theorem proving, data management, algorithmic analysis, and inferencing).

3. Language Issues. The source languages needed to specify problem solutions.

4. Environment of the environment issues. The interfacing of the environment with the underlying hardware and the interfacing of the environment with the human programmer.

5. Open Issues. A set of identified problems with no satisfactory research plans, as yet.

The remainder of this report will expand on these 5 areas.

5.1 Environmental issues.

The first, and perhaps hardest, of the issues to be discussed is the environment itself. How are objects created? transformed? interrogated? And what will be the underlying hardware which executes the system? These questions can be addressed via the following research topics.

a) Models

Research is needed in developing models to manage a system's life cycle. This report addresses one paradigm for developing software - others are possible and all should be investigated. The attractiveness of the model of section 2 is that it can be built today without many of the proposed features that will make it quite productive, yet with enough of them to allow for improvement over today's technology.

b) Transformations

The basic transformation techniques need to be investigated. The underlying assumption is that a user will develop a system in a high level requirements or specification language and the system will (semi) automatically produce transformed objects that represent the executable system. The levels of these transformations must be specified, including what sort of transformations are made. Algorithmic languages are still needed to define requirements, specifications and design. These topics will be discussed in greater detail in section 5.2 on knowledge and reasoning and in section 5.3 on languages.

c) Theoretical aspects

The current model of the software life cycle assumes requirements, design, code, test and operational phases. But actual software development rarely adheres to that pattern. For example, the operational phase consists of a "bug fixing" component and an enhancement component as new features are added to the system. This enhancement activity means a redesign, recode and retest operation. In addition conversion to new hardware is rarely an easy task. What model actually predicts this behavior?

We need a theory of program development and a quantitative measure of such development. Various complexity measures have been proposed for analyzing the structure of the source code. Individuals such as Halstead, McCabe, McCall and others have proposed measures for indicating how complex or error prone a system is likely to be [3,5]. We also need measures of design and requirements. Source program length is too restrictive a concept to apply to the entire life system, yet source code length is still the best predictor of source program errors and reliability.

Without productivity measures we have no way of knowing whether one methodology is better than another. More importantly without such measures, it is impossible for project managers to know if a given project is under control or is actually late and poorly organized. Currently lines of code per unit of time (e.g. statements per month) is the most commonly used and accurate (within a factor of 2 or 3) measure; but it is far from being acceptable. It is only reasonable during the coding phase of a project (about 2Ø per cent of the development cost) which means that the entire requirements and design phases have Ø productivity, and it does not include any of the maintenance activities where productivity, but not cost or activity, will be close to Ø. Such measures, to be effective, should be automated and ideally will be created by the transformations attached to the various objects in the environment.

d) Hardware

The effects of hardware technology of the 199Ø's will play an important part in a programming environment. Two major developments will have a major impact. These are the development of large storage systems and distributed systems with high communication bandwidths.

The first of these means that storage will be essentially free. Thus an environment will be able to save almost everything that happens. The real problem then becomes one of what to save? Much like the typical programmer (or this author), whose desk soon becomes cluttered with stacks of old computer listings, a large data base can soon become cluttered with too many retrieveable objects. The real research question is how little (not how much) of this information to save so that nothing "essential" is lost and everything "relevant" can be retrieved. Essential and relevant will be left undefined at present.

With large distributed systems (e.g. an intelligent terminal in every home connected to a central computer facility) environments will be "distributed". How to manage this large decentralized system is another major issue. On a small scale, this is related to the cross compiler - simulated environment issue in applications areas like embedded systems. If a system is developed on a large computer within one development environment, but must execute on some other computer in some other environment (e.g., in a missile or radar unit), some mechanism must be devised for either down loading the new system into the operational environment and testing it there in an effective manner or by testing the new environment within the development environment via simulation.

One possible solution is for the specifications of an object (i.e., its contract attribute) to contain details of the underlying environment. Thus the specifications of a system would be read by a simulator which would test the higher level specifications. As each object is transformed into explicit machine-oriented algorithms, the simulator at that level would have less to interpret. It would be possible with this structure to simulate other environments for testing.

Other technology issues will be discussed in the section on environment of the environment.

e) Specialization

As mentioned in section 3, the model of section 2 is designed for experts; however, there is a need to specialize the system (i.e., by restricting certain transformations) for large classes of users. Many less sophisticated programmers need to have access to the environment but may not have the knowledge to use the full range of capabilities that are available. For example, a standard programming shop might have a "canned" set of transformers for converting a specification into a source program. The programmers would simply develop the specifications and run the set of transformations.

f) Evaluation of impact

One of the hardest issues that the group tackled was one of evaluating the success of the environment. What means exist for moving the programming environment from the research laboratory into the commercial programming world? A quantitative measure of productivity could validate this environment, but none that are generally accepted currently exists, and as mentioned earlier, the development of such measures is an important part of this research area. Lines of source code seems unrealistic to consider as the basis for any such productivity measure since this environment is oriented towards producing objects and not source program listings.

It was believed that technically successful systems will generally get more use and become more popular. The example of UNIX, a generally unsupported system developed by Bell Telephone

Laboratories for the PDP 11 computer, which has become extremely popular, comes to mind. Success in the marketplace is the basic measure that we considered. Although the author is somewhat skeptical of this approach, until better quantitative measures are developed, little more can be proposed at this time.

5.2 Knowledge and Reasoning.

At a basic level, a programming environment might be thought of as an extension to present day debugging systems. However, the scope of these new environments encompasses much more. Perhaps the biggest difference is the introduction of artificial intelligence applications. There is little anticipation of large scale automatic programming systems for very complex problems; however, the computer is becoming an important logical decision making tool for use in many areas. Using computers to help make logical, yet algorithmic, decisions will enable programmers to devote more of their energies to the creative less algorithmic aspects of programming. The following areas are central to these concerns.

a) Theorem proving

The general consensus was that theorem proving techniques for algebraic languages are adequate and will not improve much. Many of the current problems with verifiers are not in the proof, but in the verification condition, i.e., what to prove by specifying the theorems to prove. The proofs are usually solvable.

There is some need to extend the domain of applicability. Pointer variables and aliasing (i.e., sharing of storage such as referring to the same location via a global variable name and a subroutine parameter) are handled by only a few systems. The extension of current verifiers to process quantificational logic is not always possible. The area of program logics is only beginning to be explored.

The major research problems still unresolved seem to be:

1. Having enough computer resources (time and space) to carry out the proofs, although the emergence of cheap easily available microprocessors may soon change that, and

2. Developing enough theories about specific problem areas, so that proofs do not always have to start from scratch. We need a "catalog" of theorems, much like the standard mathematical formulas, that can be applied to a large class of applications.

b) Algorithms

Algorithms are crucial for system operation. For increased productivity we would like to be able to reuse existing source programs, modules, algorithms, etc. The proposed transformational model permits this. A pure algorithm can be transformed into a specific instantiation usable with a specific source language.

In some applications, although code is reused from project to project, the cost savings of this reuse are not great. Significant sections of the code must be rewritten even if the "algorithm" doesn't change. Therefore, we need a mechanism for classifying such algorithms. Other than numerical functions (e.g. sine, tangent), the design of such a classification scheme has so far eluded us.

The underlying structure of the proposed transformations depends very much on algorithm design. A transformation which takes an abstract algorithm and a machine description and yields a source program on a specific machine needs as input a rich set of abstract algorithms. Thus we need to continue research in developing new basic algorithms for general purpose use. Current research in evaluating bounds on algorithm execution time or size is needed to give better cost estimates on their use.

Related to this is the need for generalized models (i.e., algorithms) of the programming environment itself. The program design methodology is also an algorithm. The current trend is to collect ideas like top down programming, stepwise refinement, walkthroughs, chief programmers, etc. and call it a methodology. These and other paradigms must be considered and evaluated.

c) Libraries and catalogues

Related to the issue of algorithm design is the need to access what is actually available. As mentioned previously, a classification scheme is needed to identify basic (non-numeric) algorithms. We need a set of standard definitions so that two algorithms that supposedly transform the same data type really do so (e.g., two sort algorithms really have the same output data given the same input data).

d) Data base design

Data bases are needed to manage the information within an environment, thus they are an important topic. Managing objects within an environment is usually a data access problem. However, the group had little to contibute to this important research area. The basic data models (hierarchical, network or relational) must be studied for applicability within this environment. Query languages needed to interrogate objects must be developed. The existence of large memories at low cost is the basic technological change we foresee in the near future.

5.3 Languages.

Language is the glue that holds the model of section 2 together. The ultimate goal of the system is to solve a given application problem by constructing an object to solve this problem. This object will specify the solution in some language. Due to the large trained programming industry and the human nature to resist change, this language will probably be some evolutionary development from current day languages. Currently Fortran and Cobol are the most widely used with languages like Pascal and PL/I used to a much lesser extent. The current interest in Ada as a new Department of Defense language for embedded computer systems shows that all language issues have not as yet been solved. While Ada is a step forward in solving many of the issues in program design (such as abstract data types and module design), there are still concepts that are not completely developed in Ada (such as real time programming, formal specifications, and parallel processing).

Current languages, such as mentioned above, are all variants of the basic Algol 60 procedural structure that is based upon the von Neumann computer architecture consisting of a large main memory and a small fast arithmetic unit. Execution proceeds by accessing a datum from the memory, manipulating it in the arithmetic unit and replacing it in the main memory. The execution is instruction by instruction. Other mechanisms are possible. Languages like LISP and APL imply a different structure and need continued investigation. Non-procedural approaches might be an important development.

This report previously addressed the need for a query language to interrogate objects. The design of such a language is also needed. The basic data contained by all objects that such a query language could access is also unknown. This query language must also interface with the programming environment's command language needed to manipulate objects and to create new objects by invoking some transformation.

Many of the problems in languages stem from the problem of separating the language from the environment. What constructs go into a language and which are given by the environment? Are digits of accuracy, array bounds and file characteristics language or environment issues? Most current languages confuse this.

The largest payoff will probably come in the area of specification languages. Current day systems are extremely simple in terms of the set of problems they can solve. There is no way to write down a complex English-oriented specification today and expect any sort of logical processing on it. There is also a need to expand the domain of applicability and generalize their usage. For example, HDM is one approach towards program verification conditions. Other approaches need be tried. Systems like PSL/PSA and SREM touch only part of the specification/requirements issues.

The basic design of a specifications language is not yet clear. In solving a large complex problem, the role of the computer must be considered. In one paradigm (e. g., SREM) the effect upon computer processing is central to the modular breakdown of the system into components. However, in others, the processing of data through the system is crucial with the computer being only a component considered along with others. There is no clear consensus as to which approach is best for a specifications language or if possibly both are required depending upon the application area. The answers are still unclear.

5.4 Environment of the Environment.

The programming environment exists as part of a larger environment - that of computers, programmers and other technical personnel. The proper use of the programming environment depends upon the proper interface between the programming environment and this larger environment.

a) Technology

Given the environment, there needs to be a mechanism for interfacing with it. One level is the programming language discussed in section 5.3. The hardware that supports the environment is another crucial factor. It was previously mentioned that the hardware we envision will consist of large memories with many dispersed systems all communicating via high bandwidth communication lines. It is assumed that computing power is "cheap." With the advent of mass production of microprocessors, that technological development has essentially arrived.

Given a distributed system, where does the environment reside? Parts will be local to the user and parts will be at some central site. How to develop such systems and to develop load sharing strategies are important research issues.

An important application area currently poorly handled is the area of embedded computer systems. In this case software for an applications computer is developed (usually) on some other larger computer. While the development computer may have testing tools for checking out the source program, the actual integration testing stage consists of very crude octal dump routines on the actual applications computer. There is an important need to integrate this environment into a programming environment. There has to be a way of simulating in an easier manner than at present the host environment on the development environment.

Compatibility between environments will be an important issue.

b) User interface

The input/output device that connects a user to an environment will greatly affect programmer productivity. Most current devices are based upon the basic teletypewriter where the user types in lines of text and the device responds with lines of text.

Graphics will have an increasingly important role in the development of future devices. The resolution on many graphics devices is increasing and the use of color adds an entire new dimension to the process. Neither development has been explored fully in the program development process. The first steps towards graphical usage are the screen editors now becoming common. Color adds information to a line of text without any loss, but has had no impact as yet on development methodologies or tools.

Better graphics devices may change the way programming languages are developed. All widely used languages assume stream input; however, a screen input may be more practical in the future. The location of a given statement on a screen can indicate its structure. Visual presentation should make it easier to comprehend a program. Thus concepts like BEGIN-END blocks may become outdated in future languages and be replaced by location constructs (e.g., prettyprinters in existing Pascal compilers), and the entire parsing and nesting structure of languages may be altered (at least at the source language level).

Voice is another communication medium that is starting to have an impact. Current systems can recognize a few words of spoken voice and some work is progressing using digitized voice for output.

Other than keyboards, touch plays little role today. However, various experimental devices have appeared from time to time and others can be expected in the future. There is a need to incorporate these devices within the Input/Output structure of existing and future programming languages.

c) Education

Educating people in the proper use of an environment is no small task. Programmers, used to a standard way of developing software, will probably resist changes such a programming environment may bring. Retraining is an issue that must be considered when an environment is developed, and not left until the very end. However, our group did not have any concrete proposals in this area. There is, also, the need to develop training programs for new programmers. In addition, the environment only specifies a set of objects and their transformations. The actual methodology to use this system must be developed.

5.5 Other Issues.

Several other topics are needed to provide an effective environment. These topics are described as follows:

a) Quality control

The ability to measure and evaluate software all through its development is critical for proper management control. This is necessary for planning a new project and for monitoring a current project to determine possible schedule slippage or other anomaly. Automating these measures with the programming environment allows for more frequent and objective measures to be collected.

Currently there is much research in complexity measures. These measures are all based upon characteristics of source programs that could be extracted by an appropriately designed compiler. A correlation between such measures and human characteristics like programming ability in a specific application area would allow for better matching of programmers and tasks. Other research in applying statistical theories of error propagation on system reliability [6] should be a useful management tool for evaluating system testing and ultimate system reliability.

Other work is seeking better measures on programmer productivity. One research problem is to simply define productivity such that it applies to the entire software life cycle. Given such a measure, management could use it to track project progress. However, in order to measure productivity, we have to collect data on what one is doing. It is not yet clear exactly what needs to be collected. All data is currently organized around the current phases of the software life cycle, yet as mentioned previously, perhaps the entire model is not the most effective.

b) Testing

Testing, a topic of interest as long as we have been programming, is still imperfect. In spite of numerous tools, the development of a random set of ad hoc test cases is still the major testing strategy. Program verifiers, data flow analyzers, symbolic execution and test data generators all exist in varying degrees of success, yet are seldom used in spite of the fact that they have been shown to be effective. More research is needed here.

c) Standards

The use of standards in programming environments must be developed carefully. Programming environments are relatively new and there are still many unanswered questions as to exactly what should be in them. To standardize too early leads to imperfect program development systems that may stifle innovation, yet to standardize too late may be futile since everyone would have his/her own variety of environment.

It seems highly unlikely that any standard environment could be developed within the time frame considered in this report. However, lower level more specific aspects of the environment are ripe for standardization. Most noticeably, protocol issues should be standardized. The interfaces between objects need clarification. There should be a standard interface between an object and its environment. Along with language standards, we certainly would want specification and requirements language standards. A common set of definitions for environmental issues is needed. The interface between a user at a terminal and the programming environment might be standardized.

5.6 What's omitted.

In describing this model, the group freely admitted that they were not experts on several topics (on most topics, but not all), or were unable to delineate clearly the benefits of the model on specific program development problems. Some of these have already been mentioned. This section contains several others.

a) System integration

It is not clear that the process of combining modules developed by separate individuals will be truly helped by this model. While we believe that it is so, we have nothing to base it on.

It is believed, however, that such a programming environment will aid in managing the interfaces between modules developed by different individuals, and thus aid in solving the system integration problem. In addition, the historical data maintained by the system for each object should aid in keeping track of a module's development history. Again this will aid in system integration.

The problem seems to be in the "bug fixing" process. Currently a programmer finds an error and fixes the problem. During this time other programmers are generally not allowed to add or change modules to the developing system. With the new model environment, the management of such modules should be enhanced, but is not clear how.

b) Maintenance

Some aspects of maintenance are aided by this model. Changes and updates to the source code can be monitored. In addition, with appropriate transformers, changes to specifications can be (semi) automatically filtered into source programs and verified.

Distribution of the altered system to many sites poses the same problems as system integration. Since each version of the developed system will execute within a different programming environment, some of the communication problems mentioned previously must be considered.

c) Evaluation and Impact

As mentioned earlier, the evaluation and impact of this new environment must be considered. Popularity is not a good measure, but is the best we came up with. The development of good objective productivity and reliability measures will aid in evaluating this model as well as many others.

6.0 References

[1] Cheatham T., J. A. Townley and G H Holloway," A system for program refinement", Fourth International Conference on Software Engineering, Munich, September, 1979, 53-62.

[2] Goldstein I. P. and D. G. Bobrow, " Extending object oriented programming in Smalltalk", 1980 LISP Conference, Palo Alto CA, August, 1980.

[3] M. Halstead, Elements of Software Science, Elsevier Computer Science Library, 1977.

[4] Ingrassia F. S.," The Unit Development Folder - a new approach to monitoring software development", Advances in Computer Programming Management, vol 1., Heyden and Son, Philadelphia, 1980, 226-238.

[5] T. McCabe, "A Complexity measure", IEEE Transactions on Software Engineering Vol 2, No. 4, 1976.

[6] Musa J. D., "A theory of software reliability and its application", IEEE Transactions on Software Engineering, Vol 1, No 3, 1975, pp 312-327.

[7] Department of Defense, Requirements for the Ada Programming Support Environment "Stoneman", February, 1980.

CHAPTER 6 SUMMARY

Group 1

Group 1 defined a series of four increasingly comprehensive, and automated environments for medium and large development projects. Each of the environments could be constructed using state-of-the-art technology. It was assumed that all environments included compilers, link-editors, assemblers, run time routines, and source coder debugging systems. For large systems, it was assumed that cross-compilers, simulators, and emulators were also available. The most modest development environment, for medium size projects, augmented the base tools with:

* a manual requirements definition and specification methodology,
* a data dictionary to facilitate design,
* an automated (but simple) source code control tool,
* a file comparator for use in verification, and
* manual milestone charts to support project management.

In all but the most modest environment, the tools interfaced with a central data base, using it as the source of information and storing results back into it. Thus the data base became the integration medium. The most elaborate environment was designed to support the development of large, real time systems. It included automated tools for requirements specification, design, extensive program analysis and testing, documentation preparation, and system management. Archiving and change control facilities for requirements, design, and source code documents were also provided. In all four of the environments the emphasis is upon organization, management, and control. Procedures are well defined and documented and records are maintained so project status is discernable throughout development. In each successive environment more of the procedures are automated and additional tools are provided to assist in analyzing and managing the products and the project itself. Group 1 felt that all four of the environments could be built today.

Group 2

Group 2 was charged with investigating environments that could be produced within 5 years. For development environments of this category, the technology for the parts and pieces is largely in place but innovation is required to integrate the components into a coherent whole. Group 2 considered five characteristics paramount for a development environment to possess:

* breath of scope and application,
* user friendliness,
* reusable components,
* tight integration, and
* use of a central information repository.

Group 2 stressed the difficulty in building a true software development environment. They felt that an environment must have a broad scope and must be based upon a deep understanding of the software development process. Although such an understanding is growing, it was felt that additional insight is required before a firm foundation for an environment can exist. Consequently a phased research approach was adopted. First, small prototype environments for specialized task areas where procedures are well defined and understood should be constructed. After experimentation with the specialized environments, broader scoped general purpose environments should be constructed. Experimentation should be directed toward the key issues associated with the critical characteristics of an environment. For example, the dependency of an environment upon life cycle models, programming language, user application, and project size should be investigated to determine the breadth of scope practical in an environment. Group 2 felt that research plans for longer than 5 years were unwise because the area is one of such creative activity that the potential for great upheaval and radical change is high.

Group 3

Group 3, tasked with defining research issues and directions for development support systems, was challenged with creative disagreement. The diversity of views underscored the ferment and activity in the research community. Basic to the conflicts were disagreements about the user and the usage setting for development environments. Some felt that 10 years in the future large development projects will still be done by large groups of people who must communicate, cooperate, and be managed. The basic tasks will remain quite similar to those of today but the development environment should provide an automated solution to many of the information and people management concerns of today. Others felt that future development projects will be accomplished in a very different manner. Application oriented users will play a greater role in the development, using languages and tools closely meshed with their application expertise. Programming environments will be so powerful and the productive potential of each developer will be so greatly increased that much smaller groups will be able to accomplish the development of even large projects. However, even with such fundamental disagreement about how software development will be done in the future, both factions agreed (more or less) upon certain technical issues.

> * Rapid prototyping---Future development systems should facilitate the quick, inexpensive creation of operational prototypes or models.
>
> * Layers of refinement-- A progressive refinement from application domain model to machine implementation was viewed as a basic mechanism for creation and representation of the product.
>
> * Documentation---Documentation should be viewed as the by-product of the production process with both the intermediate products and the process itself being saved.
>
> * Maintenance----Maintenance should be viewed as incremental redevelopment.
>
> * Information organization and management---- Organization and management of information are fundamental and critical aspects of the program development environment. Until solutions to the information management problems are found, programming environments will be unable to realize their full potential.

Group 4

Group 4 was concerned with research issues for environments based upon a programming language approach to development. They modelled an environment as a hierarchy of objects, where each object is a collection of entities. An entity is defined to be a set of attribute-value pairs. Objects are related to other objects in the hierarchy through the application of specific transformations. Program development within the environment is viewed as a series of object transformations, where each transformation refines a general or abstract object into a more detailed or concrete object. The transformations can also be used to ascertain program development information, perform optimization, produce documentation, etc. Such development environments are to serve expert users who would work in groups to develop software products. Five major areas of research activity were delineated by Group 4:

> * Environment Issues--Several key areas require investigation to develop a stronger model of programming environments: more accurate models of program development, better understanding of required object transformations, effective measuring techniques, and accurate assessment of the future hardware base.

* Knowledge and Reasoning Issues--Programming environments provide automated assistance to programmers to augment their capabilities. To do this adequately we need more knowledge about algorithms and information handling.

* Languages--A great deal of research remains to be done on languages. The inherent structure of languages and language classes needs investigation. Language approaches to specification and query are also fruitful areas for research.

* Interfaces--The influence of hardware and people on environments must have additional study and development. Utilization of future hardware technology and successful interface with users may determine the ultimate success of development environments.

* Other--How to develop high quality software is an open question that environments are being designed to answer.

Workshop Overview

As might be expected in a new area of high interest, the workshop participants did not speak with one voice. However, there was agreement on two major items:

* the importance of using automation to assist software development, and

* that successful information management was a critical but perplexing issue.

The group tasked with the most near term approach suffered the least disagreement and developed the most definite statement. The groups tasked with looking into the future found quite a range of images in their respective crystal balls. Perhaps the area of most fundamental disagreement was the conception of what could be accomplished within a 5 io 10 year timeframe. It appeared that this disagreement was based upon very different perceptions of the current state-of-the-art. Some saw present day reality in terms of projects underway in the research labs; others viewed reality as mirrored by the technology in use in current production shops. The profound lag in technology transfer from construction of research prototypes to the use of the concept in the production arena impacted the workshop as it does the industry. Another broad area of disagreement was the intended user group for future software development environments. The disparity of views about the user population led to differences of opinion about the functions required within a development environment. Considering the pervasiveness of the disagreement, there was surprising agreement about technical approaches. Software development was viewed as a series of refinements of objects from the general requirement specification to the concrete realization of the program. Discovery of the transformations and the increasingly automated application of the transformations was seen as an important research topic. The need for the rapid construction of prototypes was viewed as an important component of future development environments and an important concept to apply to the development of the environments themselves. It was felt that only through continued environment construction, experimentation, and reconstruction would the goal be achieved. Almost everyone saw a data repository as the heart of a software development environment. However, there were few opinions expressed on how to deal with the wealth of information that all felt should be kept. The participants agreed that development support systems and programming environments were important topics and ripe research areas.

Omissions

Although many topics were discussed at the workshop and many issues raised, in reviewing the results, major omissions became evident.

1. Identification of major settings.

During workshop planning sessions it was agreed that the type of programming environment required would be highly dependent upon the setting in which it was to be used. Identification of major settings was deemed a task that merited attention at the workshop, but it was never adequately addressed.

2. Determination of what to do with the wealth of information comprising the environment data repository.

There was almost universal agreement that a data repository was at the heart of a programming environment or development support system, but little more was said on the subject. What information should be saved? How it should be organized? Who should manage or control the information? To whom should what access be granted? All are questions that if even asked, generated few responses. Handling of the data associated with an advanced development support system is a key issue that may well determine the ultimate success of such systems.

3. Validation of programming environment technology.

The entire software field suffers from an inability or perhaps an unwillingness to validate its technology. In the beginning of this still rather new field, the mere existence of a working system (or program) that seemed to accomplish its designated task, was taken as proof enough of the validity of the approach. As scientists become more concerned with which of several proposed ways is the best, the question of validation creates problems. Projects are so expensive that it is not feasible to perform experiments of realistic size that compare alternative techniques for development. Possible approaches to this problem of determining the validity of techniques was sought at the workshop. No discussions ensued.

Recommendations for NBS

Obviously it is too early to specify detailed standards for sophisticated software development support systems and development environments. However, it is not too soon to consider the standardization issues and approaches. Inopportune environment design could preclude potentially advantageous approaches to standardarization. The most promising and appealing approach is to define standard features, protocols and interfaces so that functional pieces can be independently developed and yet be compatible and capable of being used as building blocks in the construction of environments. The workshop suggested specific avenues to pursue.

APPENDIX A

Group 1: Contemporary Systems

Paul Cohen, DCA
William Howden, UC-San Diego
Al Irvine, Softech
James King, IBM
Patricia Powell, NBS
William Riddle, Cray Labs
Leon Stucki, BCS
Leonard Tripp, BCS

Group 2: Five Year Systems

Lori Clarke, U. of Massachusetts
Donald Good, U. of Texas
Raymond Houghton, NBS
Thomas Love, ITT
Leon Osterweil, U. of Colorado
Patricia Santoni, NOSC
Daniel Teichroew, U. of Michigan
Anthony Wasserman, UC-San Francisco

Group 3: Support Systems

Robert Balzer, ISI
David Barstow, Schlumberger-Doll
Meera Blattner, LLL
Martha Branstad, NBS
John Buxton, U. of Warick
Adele Goldberg, Xerox
Robert Morris, Bell Labs
Stephen Squires, NSA
Thomas Standish, UC-Irvine

Group 4: HLL Approach

W. Richards Adrion, NSF
Alfred Aho, Bell Labs
Daniel Bobrow, Xerox
Thomas Cheatham, Harvard
John Cherniavsky, NSF
Susan Gerhart, ISI
Gordon Lyon, NBS
John Nestor, Carnegie-Mellon
Terry Straeter, General Dynamics
Marvin Zelkowitz, U. of Maryland

APPENDIX B

QUESTIONS AND ISSUES

In order to clarify the domain of each working group and stimulate thought, a list of questions and issues has been prepared. A set of general questions plus sets specific to each working group have been included.

General Questions

1. Support requirements seem to vary drastically depending upon the application, project size, general skill level of the developers, etc. Identify major categories of environment settings. For example, one category is the single gifted user setting.

2. Given the setting, what are the requirements?

3. What is the impact of using a programming environment, and of not using one? How can the impact be determined?

4. What criteria should be used for determining the suitability of a programming environment?

5. How can the technology be validated?

6. What is the state of the art?.
   - What can we do today?
   - What can we do starting today?
   - What should we plan for tomorrow?

Group 1: Tools and Techniques

Assumptions: Today's technology

Group products:
   1. list of core tool functions
   2. discussion of feasibility of standard tools
   3. discussion of mechanisms for standardizing tools
   4. discussion of tool research issues

Issues:

1. Identification of the core set of tool functions and systematic development techniques.

   * what tool functions do various classes of development personnel require?
   * what tool functions support which life cycle activities?
   * what tool functions support which life cycle products?

2. Identify tool properties and discuss their importance.

   * availability
   * capital costs
   * operating costs
   * functional capabilities
   * complexity
   * life cycle placement
   * automation
   * effectiveness
   * domain and range
   * auditing
   * management
   * applicability
     . area of application
     . language
     . project size

3. Tool organization and availability

   * identify those tool functions currently available as features of automated development tools and classify those tools as multi- or single function
   * indicate those not currently available, but which can be constructed with today's technology as extensions to current tools or as new tools.
   * compare the advantages of large multifunction tools with those for collections of independent single function tools

Group 2: Integrated Systems

Assumptions: today's technology with a 5 year delivery date

Group products:
1. feasibility of standard integrated system
   a. standard framework plus tools as building blocks
   b. standard system for each Federal standard language
   c. standard tool interfaces
2. mechanisms for handling a standard integrated facility

Issues:

1. Is there an ideal or best methodology for the software development process? What are desirable features of development methodologies?
   * What are the principal activities during development?
   * What data flows between these activities?
   * What metrics can be used to decide when to go to another activity?
   * What automation exists for these activities?
   * What automation has been the most effective?
   * What information is needed to support these activities?
   * What are the proper roles and relations of human choice and machine support?
   * What are the most critical activities and what support could most improve performance in these areas?

2. What are the primary products produced during software development?
   * What are they called and what is their nature?
   * What information content should they have?
   * How do they feed each other?
   * How, when, and by whom are they produced?
   * What tools are needed to facilitate the production, storage, and retrieval of these products?

3. How should tools be organized?
   * What development activities should be supported?
   * How well are related activities supported by related tools?
   * Indicate data inputs/outputs for tool classes.
   * To what extent can existing tools be integrated and at what cost?
   * Major design issues
     . interfaces
     . representation
     . data management
     . extensibility
   * List tool areas that need further development.
   * Which methodologies can be supported by currently available tools?

Group 3: Advanced Support Systems

Assumption: research and development area

Products:
1. list of key technical issues
2. framework for critical research and development
3. feasibility of standard development support systems

Questions and Issues:

1. Design Goals: What should the design goals of an advanced development support system be? What purposes should an advanced programming environment serve?

2. Software Quality: What aspects of software quality should an advanced programming environment support: (a) in the advanced environment itself?, and (b) in the application systems the environment is used to construct and maintain?

3. Software Lifecycle: What phases of the software lifecycle should an advanced programming environment support? (a) requirements analysis, (b) specification, (c) design, (e) implementation and coding, (f) debugging, testing, and module integration, (f) maintenance and upgrade?

4. Software Management Support: What software management support should an advanced programming environment provide? (a) cost estimation, (b) critical path scheduling, (c) task schedule management, (d) reporting on resources spent, (e) monitoring of performance of programmer teams, (f) others.

5. Program Development Tools: What program development tools should an advanced programming environment support? (a) program analysis, (b) program transformation and optimization, (c) text and document management, (d) version control, (e) others.

6. Database Properties: What should be the structure of the database for an advanced programming environment? What should it contain? What policies should there be for user access privileges and capabilities?

7. Documentation: What special support, if any, should be given to documentation in an advanced programming environment?

8. User Interface: What should be the properties of the user interface to an advanced programming environment? (a) consistent? (b) extensible (as with the UNIX Shell)?, (c) others?

9. Pervasive Services: What pervasive services should an advanced programming environment offer (i.e. services available at any time that can interrupt any activity, and which allow the interrupted activity to resume after use)? (a) help system, (b) on-line manuals and documentation, (c) others?

1Ø. Maintenance and Upgrade: What special tools and disciplines should an advanced environment support, if any, for the activities of maintenance and upgrade?

Group 4: Language Environments

Assumption: research area

Products:

1. feasibility of approach and domain of applicability
2. major research issues
3. timeframe for research and development
4. impact on and interaction with Federal standards

Questions:

1.Language issues in a HLL programming environment

*Language features

- Can a consistent and complementary language be used to describe requirements, specification, design, code, testing and maintenance?
- What will such a language look like? Will it be procedural (e. g. Algol-like), applicative (e.g. Lisp-like) or other?
- Will we need several 'similar' languages to use over a project's lifetime? Do we now have the technology to build such a system?
- What concepts now under study will be more important for new languages?
- What ideas not now being studied should be?
- Some of the concepts now being investigated include: encapsulated data types, modules and information hiding, top down, bottom up, stepwise refinement, stubs, threads, etc., process protection and synchronization, formal specifications, source code verification, testing strategies, What should be added to this list, and what deleted?

*The concept of a program design language (PDL) or pseudo code has been proposed as a design language.

- How can these be automated?
- What information can such a compiler produce to aid the designer "see" the entire design?
- How can these designs be converted to source code?
- How can these be automatically updated whenever source code is updated?
- Can this idea be extended to specification languages or requirement languages?

2. Interface with other technologies

*What sort of tools are needed?

- Are the current crop of tools sufficient, or do we need to rethink the entire development process?
- Most current tools emphasize development. How do we

incorporate maintenance and enhancement into process (and language)?

*What effects will the current hardware technology have on language environments; specifically, what will cheap microprocessors with large memories have?

*Integrated tools will need to access a project data base.
. What sort of language is needed?
. How do we manage the insertion and retrieval of data?

*Does artificial intelligence play a role here?
. What sort of knowledge-based data structures are needed?
. Do we have the technology today to build such a system?

3. Impact and directions

*How do we measure productivity?
. What sort of accounting, data collection and auditing is needed in such an environment?

*What are the current major efforts in this area today?

*Where should research (and research funds) be directed over the next 5 jo 1Ø years?

*Can we identify who's now doing what?

☆ U.S. GOVERNMENT PRINTING OFFICE : 1981 O—340-997 (1719)

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS SP 500-78	2. Performing Organ. Report No.	3. Publication Date June 1981

4. TITLE AND SUBTITLE

NBS Programming Environment Workshop Report

5. AUTHOR(S)

Martha A. Branstad and W. Richards Adrion, Editors

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as item 6.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 81-600068

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

In May of 1980, NBS hosted a workshop to assess the state-of-the-art in programming environment technology and to determine the key questions and issues that must be addressed to use these techniques to improve software quality and productivity within the Federal Government. This document reports the results of the workshop.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

development support systems; programming environments; software development; software tools; toolboxes.

13. AVAILABILITY

[✓] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
[✓] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

106

15. Price

$4.75

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, D. C. 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name ____

Company ____

Address ____

City ____________________ State ______________ Zip Code __________

(Notification key N-503)